# Compilation and Backend-Independent Vectorization for Multi-Party Computation

Anonymous Author(s)

## ABSTRACT

Research on MPC programming technology largely falls at the two ends of the classical compiler: (1) work on front-end language design (e.g., Wysteria, Viaduct) and (2) work on back-end protocol implementation (e.g., ABY, MOTION).

In this work, we formalize the MPC Source intermediate language and advance what we call *backend-independent* optimizations, in a close analogy to machine-independent optimizations in the classical compiler. We present a compiler framework that takes a Python-like routine and produces MOTION code. We focus on a specific backend-independent optimization: novel SIMD-vectorization on MPC Source, which we show leads to significant improvement in circuit generation time, running time, and communication over the corresponding iterative schedule.

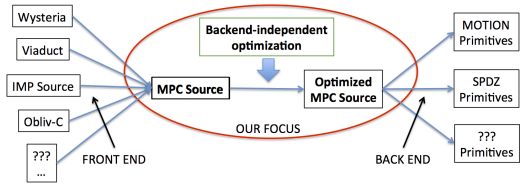
## 1 INTRODUCTION

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 improve technology so that 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.

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 formalize the MPC Source [?] intermediate representation and emphasize optimization over MPC Source. As in classical compilers, we envision different front ends (e.g., our front end IMP Source) 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 analysis and optimizations that take into account control and data flow in a specific program. 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 Source language, its syntax and semantic restrictions, (b) translation 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 demonstrates high performance [?]. Second, it provides an API over efficient implementation for a wide variety of cryptographic operations (e.g., MUL, CMP, etc.) 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 (e.g., MUL_SIMD), which amortize cost and lead to significant improvement in memory footprint and throughput [?, ?, ?]. It enables MPC Source-level vectorization, a key focus of this paper.

Our second contribution is an analytical model for cost estimation of amortized schedules. Originally, we hoped that

optimal scheduling (under our model, which essentially minimizes the length of the schedule) 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 modeling is important as it drives not only vectorization but optimizations such as protocol mixing 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 16 programs with interleaved if- and for-statements; these include classical MPC benchmarks such as PSI and Biometric matching, as well as kMeans, Histogram, and other examples from the literature. Our compiler takes the routine and generated *non-vectorized* MOTION code (from MPC Source on the picture above). It then optimizes MPC Source and generates *vectorized* MOTION code (from Optimized MPC Source). We then run the two versions using Boolean GMW and the BMR protocols. (MOTION, which is designed for protocol mixing, supports Arithmetic GMW, however, it does not implement Comparison (CMP) and Multiplexing (MUX) as they would be rather inefficient.) In our LAN experiments vectorized code exhibits 24x improvement on average in circuit generation time for Boolean GMW (20x for BMR), 7x reduction in communication (2x), 97x reduction in number of gates (91x), 4x improvement in setup time (23x) and 21x improvement in online time (18x).

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, ushering new MPC optimizations and combinations of optimizations in the vein of standard compilers, and thus bringing 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. §**??** details the front-end phases of the compiler, §**??** focuses in on backend-independent vectorization, and §**??** describes translation into MOTION. §**??** presents the experimental evaluation.§**??** discusses related work and §**??** concludes.

All our code, including benchmark Python-like code, compilation phases, and generated MOTION code is available on Github. We omit the link for anonymity, however, we will gladly make it available upon request from reviewers. The Github setup generates graphs and intermediate code for each benchmark along each compiler phase, it compiles with MOTION and runs the circuits on small inputs to generate tables of data (the experiments we present later run on real LAN and WAN). We plan to release the link and code, which we believe will be useful to researchers.

## 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 [**?**] syntax and imposes certain semantic restrictions. The programmer writes an iterative program and annotates 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.

### 2.2 MPC Source and Cost of Schedule

Our compiler generates an intermediate representation, MPC Source. MPC Source is a *linear* SSA form. MPC Source for Biometric Matching is shown and described in detail in Listing **??**(b).

We turn to our analytical model to compute the *cost* of the iterative program. Assume cost $\beta$ for a local MPC operation (e.g., ADD in Arithmetic sharing) and cost $\alpha$ for a remote MPC operation (e.g., MUX, CMP, etc.). Assuming that ADD is $\beta$ and SUB, CMP and MUX are $\alpha$, the MPC Source in Listing **??**(b) gives rise to an iterative schedule with cost $ND(2\alpha + \beta) + N(3\alpha)$.

A key contribution is the vectorizing transformation. We can compute all $N * D$ subtractions (line 9 in (b)) in a single SIMD instruction; similarly we can compute all multiplications (line 10) in a single SIMD instruction. And while computation of an individual sum remains sequential, we can compute the $N$ sums in parallel.

### 2.3 Vectorized MPC Source and Cost of Schedule

Our compiler produces the vectorized program shown and described 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.

In MPC back ends, 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 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 $\beta$-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

```
1  def biometric(C: shared[list[int]], D: int,
2          S: shared[list[int]], N: int) ->
3          shared[tuple[int,int]]:
4      min_sum : int = MAX_INT
5      min_idx : int = 0
6      for i in range(N):
7          sum : int = 0
8          for j in range(D):
9              # d = S[i,j] - C[j]
10             d : int = S[i * D + j] - C[j]
11             p : int = d * d
12             sum = sum + p
13         if sum < min_sum:
14             min_sum : int = sum
15             min_idx : int = 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 (a) IMP Source to (b) MPC Source: First, MPC Source is an SSA form. Second, it is linear. The conditional in lines 13-15 in IMP Source turns into the linear code in lines 12-16 in MPC Source. The test turns into the 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 $\phi$-nodes at the if-then-else (lines 13-15) turn into MUX operations, while the $\phi$-nodes at for-loops turn into *pseudo* PHI nodes with a straightforward semantics. ==== From (b) MPC Source to (c) Optimized MPC Source: 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.**

both the vectorized and non-vectorized programs.) The first term in the sum corresponds to the vectorized subtraction and multiplication (lines 9-10 in (c)), the second term corresponds to the for-loop on $j$ (lines 12-16) and the third one corresponds to the remaining for-loops on $i$ (lines 19-25). Clearly, $2\alpha_{fix} + D\beta_{fix} + N(3\alpha_{fix}) << ND(2\alpha_{fix} + \beta_{fix}) + N3\alpha_{fix}$. Empirically, we observe that (1) $\alpha_{var} \approx 0$ and (2) there is orders of magnitude improvement in running time and memory. E.g., we see about 12x improvement in online time in GMW for $N = 128$. Additionally, the non-vectorized version runs out of memory for $N = 256$, while the vectorized one runs with the standard maximal input size $N = 4,096$ (and perhaps larger $N$).

## 3 ANALYTICAL MODEL

This section presents a model to reason about the cost of execution of MPC programs, including accounting for amortization. We define the assumptions and setting in §**??**. We proceed to define the scheduling problem in §**??**, which we expected to be able to solve optimally. §**??** shows that the problem is NP-hard via a reduction to the Shortest Common Supersequence (SCS) problem. Despite the negative general result, we expect the formulation in terms of SCS to be useful as sequences are short and few in practice.

### 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 (e.g., ADD in Arithmetic, XOR in Boolean) has cost $\beta$ and a remote instruction (e.g., MUX, MUL, SHL, etc.) has cost $\alpha$, where $\alpha >> \beta$. We assume that all remote instructions have the same cost.
(3) Following Amdahl's law, 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 preparing and sending a packet from party 0 to party 1 for example.) Therefore, amortized execution of $n$ operations is $f(n) = \alpha_{fix} + n\alpha_{var}$ in contrast to unamortized execution $g(n) = n\alpha_{fix} + n\alpha_{var}$. We have $\alpha_{fix} << n\alpha_{fix}$ and since fixed cost dominates variable cost (particularly for remote operations), we have $f(n) << g(n)$.
(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".[1]

### 3.2 Problem Statement

As mentioned earlier, at the lowest level, we have two types of MPC instructions (also called *gates* or *operations* in similar works) 1) local/non-interactive (e.g., an addition instruction $A$) and 2) remote/interactive (e.g., 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 $P_i$'s consist of MPC instructions of the same kind, e.g., all MUL, MUX, ADD, etc.
(2) Def-use dependencies of the graph $G(V, E)$ are 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 $i' < j'$.

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

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

The cost of schedule $S$ is

$$cost(S) = \sum_{i=1}^{n} cost(S_i) = L_\alpha\alpha_{fix} + L_\beta\beta_{fix} + L_\alpha\alpha_{var} + L_\beta\beta_{var} \quad (1)$$

where $L_\alpha$ is the number of $\alpha$-instructions and $L_\beta$ is the number of $\beta$ ones. (We used this formula to compute the cost of the unrolled MPC Source program in §**??**.) The cost of $P$ is more interesting:

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

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 $|P_i|$ $\alpha$-instructions, or $cost(P_i) = \beta_{fix} + |P_i|\beta_{var}$ if it stores $\beta$-instructions. (Similarly, we used this formula to compute the cost of the Optimized MPC Source program in §**??**.)

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 was tractable. Unfortunately, the optimal schedule turns out to be NP-hard via a reduction to the Shortest Common Supersequence problem.

### 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 $\{s_1, \ldots, s_n\}$ of operations. In each sequence $s_i$ operations depend on previous operations via a def-use i.e. $s_i[j], j > 1$ depends on $s_i[j-1]$.

As an example, consider the MPC program consisting of the following three sequences, all made up of two distinct $\alpha$-instructions $M_1$ and $M_2$, e.g., $M_1$ is MUL and $M_2$ is MUX. 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 shortest common supersequence problem [**?**] is as follows: *given two or more sequences find 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. We can 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 $P$ is a supersequence of each sequence $s_i$, i.e., $P$ is a common supersequence of $s_1 \dots s_n$. 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 $L'\alpha_{var} + N\alpha_{var} < L\alpha_{var} + N\alpha_{var}$, contradicting the assumption that $P$ has the lowest cost. □

# 4 COMPILER FRONT END

Fig. **??** presents an overview of our compiler. This section outlines the front end, §**??** describes analysis on MPC Source and backend-independent vectorization and §**??** outlines the back end. Details on all parts appear in [**?**].

## 4.1 Syntax and Semantic Restrictions

Source syntax is essentially standard IMP syntax but with for-loops:

| | |
|---|---|
| $e ::= e\ op\ e \mid \texttt{x} \mid \texttt{const} \mid \texttt{A[}e\texttt{]}$ | *expression* |
| $s ::= s; s \mid$ | *sequence* |
| $\texttt{x} = e \mid \texttt{A[}e\texttt{]} = e \mid$ | *assignment stmt* |
| $\texttt{for}\ i\ \texttt{in range(}I\texttt{)} : s \mid$ | *for stmt* |
| $\texttt{if}\ e\texttt{:}\ s\ \texttt{else:}\ s$ | *if stmt* |

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

| | |
|---|---|
| $\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{list[int]} \mid \texttt{list[bool]}$ | *base types* |
| $q ::= \texttt{shared} \mid \texttt{plain}$ | *qualifiers* |

The type system is mostly standard, and in our experience, a sweet spot between readability and expressivity. The `shared` qualifier denotes shared values, i.e., ones shared among the parties and computed upon under secure computation protocols; the `plain` qualifier denotes plaintext values. Subtyping is `plain` <: `shared`, meaning that we can convert a plaintext value into a shared one, but not vice versa. Subtyping on qualified types is again as expected, it is covariant in the qualifier and invariant in the type: $\langle q_1\ \tau_1 \rangle <: \langle q_2\ \tau_2 \rangle$ iff $q_1 <: q_2$ and $\tau_1 = \tau_2$.

Our compiler imposes certain semantic restrictions that it enforces throughout the various phases of compilation. We note that in some cases, the restrictions can be easily lifted and we plan to do so in future iterations of the work.

(1) Loops are of the form $0 <= i < I$ and bounds are fixed at compile time. It is a standard restriction in MPC that computation is bounded [**?**, **?**].
(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 as specified by the rule:

$$\frac{\Gamma \vdash e : \langle \texttt{plain int} \rangle \quad \Gamma \vdash \texttt{A} : \langle q\ \texttt{list}[\tau] \rangle \quad \tau \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash \texttt{A[}e\texttt{]} : \langle q\ \tau \rangle} \text{ (Array Access)}$$

The subscript $e$ is a function of the indices of the enclosing loops. For read access, the compiler allows an arbitrary such function. However, it restricts write access to *canonical writes*, i.e., $\mathsf{A}[i, j, k] = \dots$ where $i$, $j$ and $k$ loop over the three dimensions of A. Write access such as for example $\mathsf{A}[i, j+2] = \dots$ is not allowed. (This is again a restriction we imposed for convenience in our current implementation; we plan to extend the compiler with arbitrary write access.)

For the rest of this section 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. 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 $i, j, k$, however, discussion easily generalizes to arbitrarily large loop nests.

## 4.2 From IMP Source to SSA

Our compiler translates from Source to SSA as sketched below.

*Parsing:* Use Python's `ast` module to parse the input source code to a Python AST.

*Syntax checking:* Ensure that the AST matches the restricted subset defined in Section §**??**.

*3-address CFG conversion:* Convert the restricted-syntax AST to a three-address control-flow graph (CFG). The step processes for-loops, if-statements and assignments as restricted by the syntax.

*SSA conversion:* Convert 3-address CFG to SSA with Cytron's algorithm [**?**].

## 4.3 From SSA to MPC Source

Once the compiler converts the code to SSA, it transforms $\phi$-nodes that correspond to if-statements into MUX nodes. From the 3-address CFG conversion step, $\phi$-nodes corresponding to if-statements will be in a basic block with the "merge condition" property. E.g., if $\mathsf{X!3} = \phi(\mathsf{X!1}, \mathsf{X!2})$ is in a block with merge condition C, the compiler transforms it into $\mathsf{X!3} = \mathsf{MUX(C, X!1, X!2)}$. Next, the compiler runs the dead code elimination algorithm from [**?**].

Next, 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

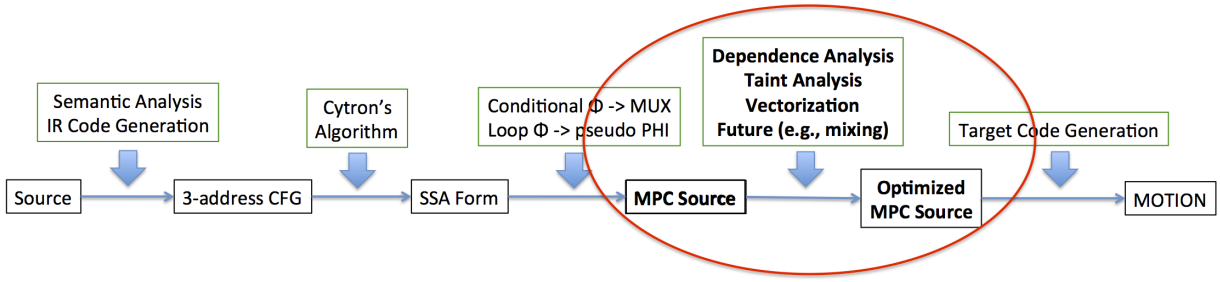



**Figure 1: Compiler Framework.**

both branches of the if-statement are visited. (The Python AST naturally gives rise to a translation where each conditional has exactly two targets, and each "merge condition" block has exactly two incoming edges, a TRUE and a FALSE edge. Thus, each $\phi$-node has exactly two multiplexer arguments, which dovetails into MUX. This is in contrast with Cytron's algorithm which operates at the level of the CFG and allows for $\phi$-nodes with multiple arguments.) 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 to the main MPC source.

Now, the remaining $\phi$-nodes in MPC source are the loop header nodes. We call these nodes *pseudo* $\phi$-nodes and we write PHI in MPC Source. A pseudo $\phi$-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 $\phi$-node evaluates to X!0, otherwise, it evaluates to X!2.

# 5 BACKEND-INDEPENDENT VECTORIZATION

This section describes our vectorization algorithm. While vectorization is a longstanding problem, and we build upon existing work on scalar expansion and classical loop vectorization [**?**], our algorithm is unique as it works on the MPC Source SSA-form representation. We posit that vectorization over MPC Source is a new problem that warrants a new look, in part because of MPC's unique linear structure and in part because vectorization meshes in with other MPC-specific optimizations in non-trivial ways (other works have explored manual vectorization and protocol mixing in ad-hoc ways, e.g., [**?**]).

In this section, we briefly describe the analysis. Details, including edge cases and examples, can be found in the extended version [**?**].

## 5.1 Dependence Analysis

We build a dependence graph where the nodes are the MPC Source statements and the edges represent the def-use relations.

*Def-use Edges.* We distinguish 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 9 to 10 in the Biometric MPC Source in Listing **??**(b) is a same-level 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 $i, j, k$. E.g., 1 to 4 in Biometric forms is an outer-to-inner edge.
- inner-to-outer $X \to Y$ where $X$ is in an inner loop nest, $i, j, k$, and $Y$ is in the enclosing loop nest $i, j$. E.g., the def-use from 8 to 12 gives rise to an inner-to-outer edge.
- 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 $\mathsf{x}' = \mathsf{x}$ immediately after the $i, j, k$ loop. Then we replace the use of x at $Y$ with $\mathsf{x}'$. This transforms a mixed forward edge into an "inner-to-outer" forward edge followed by an outer-to-inner forward edge.

*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*$(n)$.

## 5.2 Scalar Expansion

An important component of our algorithm is expansion of scalars and arrays to the corresponding loop dimensionality, which is necessary to expose opportunities for vectorization. In the Biometric 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 scalar d is lifted: d[i,j] = S[i,j] - C[i,j]. The algorithm then detects that the statement can be vectorized.

Expansion (and also reduction) is done with the *raise_dim* and *drop_dim* functions, which we believe are standard. *raise_dim* takes the original array, the access function $f(i,j,k)$ in loop nest $i,j,k$ and the loop bounds $((i:I),(j:J),(k:K))$ and produces a new 3-dimensional array $\mathsf{A}'$ by iterating over $i,j,k$ and setting each element of $\mathsf{A}'$:

$$raise\_dim(A, f(i,j,k), ((i:I),(j:J),(k:K))) : \mathsf{A}'[i,j,k] = \mathsf{A}[f(i,j,k)]$$

Raise dimension applies when reshaping input arrays and also, at outer-to-inner def-use edges. Analogously *drop_dim* applies at inner-to-outer def-use edges. It takes a higher dimensional array, say $i,j,k$ and removes trailing dimensions, say $j,k$. 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 $k$:

$$drop\_dim(\mathsf{A}, (j:J,k:K)) : \mathsf{A}'[i] = A[i, J-1, K-1]$$

### 5.3 Basic Vectorization

There are two key phases of the algorithm. Phase 1 inserts raise dimension and drop dimension operations according to def-uses. E.g., if there is an inner-to-outer dependence, it inserts *raise_dim*, and similarly, if there is an outer-to-inner dependence, it inserts *drop_dim*. For example, lines 4,5 in Biometric **??**(c) reshape the input arrays. After this phase, operations work on arrays of the corresponding dimensionality and we optimistically vectorize all arrays.

Phase 2 proceeds from the inner-most towards the outer-most loop. For each loop it anchors dependence cycles (closures) around pseudo PHI nodes then removes vectorization from the dimension of that loop. In the Biometric example **??**(b), the iteration on the inner loop (7-11) determines that $j$ is not vectorizable. The next iteration on the outer loop determines that $i$ is vectorizable. It also splits the remainder of the loop anchoring each loop around the corresponding PHI node.

There are two important points in this phase. First, it may break a loop into smaller loops which would discover opportunities for vectorization in intermediate statements in the loop. Second, it handles writes arrays. It creates opportunities for vectorization in the presence of write arrays, even though Cytron's SSA adds a back-edge to the array PHI-node, thus killing vectorization of statements that read and write that array.

The excerpts in red color in the pseudo code below highlight the extension with array writes. We advise the reader to omit the extension for now and consider just read-only arrays. We explain the extension in §**??**. (As many of our benchmarks include write arrays, it plays an important role.)

Phases 3 reverses scalar expansion (this is an optional phase and our current implementation does not include it).

*Pseudocode:*

{ Phase 1: Raise dimension of scalar variables to corresponding loop nest. We can traverse stmts linearly in MPC-source. }
**for** each MPC $stmt : \mathsf{x} = Op(\mathsf{y}_1, \mathsf{y}_2)$ in loop $i,j,k$ **do**
  **for** each argument $\mathsf{y}_n$ **do**
    case def-use edge $stmt'$(def of $\mathsf{y}_n$) → $stmt$(def of x) of
      same-level: $\mathsf{y}'_n$ is $\mathsf{y}_n$
      outer-to-inner: add $\mathsf{y}'_n[i,j,k] = raise\_dim(\mathsf{y}_n)$ at $stmt'$
      (more precisely, right after $stmt'$)
      inner-to-outer: add $\mathsf{y}'_n[i,j,k] = drop\_dim(\mathsf{y}_n)$ at $stmt$
      (more precisely, in loop of $stmt$ right after loop of $stmt'$)
  **end for**
  { Optimistically vectorize all. $I$ means vectorized dimension. }
  change to $\mathsf{x}[I,J,K] = Op(\mathsf{y}'_1[I,J,K], \mathsf{y}'_2[I,J,K])$
**end for**
{ Phase 2: Recreating for-loops for cycles; vectorizable stmts hoisted up. }
**for** each dimension $d$ from highest to 0 **do**
  **for** each PHI-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` $i_d$ `in` ... loop { i.e., MOTION loop }
    add PHI-nodes in $cl$ to header block
    add target-less PHI-node for A if $cl$ updates array A
    add statements in $cl$ to loop 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 for def-uses: e.g., some def-use edges become same-level.
  **end for**
  **for** each target-less PHI-node A!1 = PHI(A!0,A!k) **do**
    in vectorizable stmts, replace use of A!1 with A!0
    discard PHI-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` $i$ ... MOTION loop end at target (use) $\mathsf{x}' = drop\_dim(\mathsf{x}, i)$. }
**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 $\mathsf{x}' = drop\_dim(\mathsf{x}, i)$ **do**
  replace with $\mathsf{x}' = \mathsf{x}$ 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

## 5.4 Correctness Argument

We build a correctness argument that loosely follows Abstract Interpretation. First we define the MPC Source syntax. 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* (as in §**??**), 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 $C$. The theorems state (informally) that the vectorized program preserves def-uses and thus computes the same result as the original program.

*MPC Source Syntax.* Fig. **??** states the syntax and linearization semantics of MPC Source. Although notation is heavy, the linearization simply produces schedules as discussed in §**??**; the iterative MPC Source gives rise to what we called sequential schedule where loops are unrolled and MPC Source with vectorized dimensions gives rise to what we called parallel schedule. For simplicity, we consider only scalars and read-only arrays, however, the treatment extends to write arrays as well. $\mathsf{x}[i, J, k]$ denotes the value of scalar variable x at loop nest $i, j, k$. Upper case $J$ denotes a vectorized dimension and lower case $i, k$ denote iterative dimensions. Our compiler imposes semantic restrictions over the syntax: (1) x is treated as a 3-dimensional array and (2) $\mathsf{x}[i, J, k]$ must be enclosed into for-loops on non-vectorized dimensions $i$ and $k$:

```
for i in range(I):
    for k in range(K):
        ... x[i,J,k] ...
```

*Partial Orders.* For each MPC Source program $a$ we compute the def-use edges in the standard way: if statement $s1 \in a$ defines variable x, e.g., $\mathsf{x}[i, j, k] = ...$, and statement $s2 \in a$ uses x, e.g., $... = ...\mathsf{x}[i, j, k]$ and there is a path in 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 statements, $s \in a_0$ and $s' \in a_1$ are *same*, written $s \equiv s'$ 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., $s[i, j, k] \equiv s'[I, j, K]$. We extend the definition to def-use edges in the obvious way: a def-use edge $s_0 \to s_1$ in $a_0$ and an edge $s'_0 \to s'_1$ in $a_1$ are *same*, written $s_0 \to s_1 \equiv s'_0 \to s'_1$, if and only if $s_0 \equiv s'_0$, $s_1 \equiv s'_1$, and the two edges are both either forward or backward.

Definition 1. *Let $a_0, a_1 \in A$. We say that $a_0 \leq a_1$ iff for every def-use edge $e$ in $a_0$ there is an edge $e'$ in $a_1$ such that $e \equiv e'$.*

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.* The two theorems arising from the Basic vectorization optimization are as follows:

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

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

These theorems simply state that the transformation preserves def-use relations which is an invariant of the algorithm.

## 5.5 Extension with Array Writes

Array writes limit vectorization as they sometimes introduce infeasible loop-carried dependencies. For example, consider one of our benchmarks, Histogram. The source takes two arrays A and B of size N, where $\mathsf{A}[j]$ stores the bin number and $\mathsf{B}[j]$ stores the corresponding value that we need to add. The MPC Source is:

```
for i in range(0, num_bins):
  res1 = PHI(res, res2) # res is the result array
  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
```

There is a def-use edge to 2 due to the update at 8 of array res even though the loop can be vectorized across $i$.

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 $i, j, k$. Consider a loop $j$ enclosed in some fixed $\underline{i}$. Only if an update (definition) $\mathsf{A}_m[f(i, j, k)] = ...$ at some iteration $\underline{j}$ references the *same* array element as a use $... = \mathsf{A}_n[f'(\underline{i}, j, k)]$ 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, shown in [**?**], 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 $f(\underline{i}, \underline{j}, \underline{k}) = f'(\underline{i}, \underline{j'}, \underline{k'})$ (using

| | | | |
|---|---|---|---|
| $s$ | $::= s_1; s_2$ | $\gamma(s) = \gamma(s_1)\ ;\ \gamma(s_2)$ | *sequence* |
| | $\mid \mathsf{x}[i,J,k] = \mathsf{op\_SIMD}(\mathsf{y}_1[i,J,k], \mathsf{y}_2[i,J,k])$ | $\gamma(\mathsf{x}[i,J,k] = \mathsf{op\_SIMD}(\mathsf{y}_1[i,J,k], \mathsf{y}_2[i,J,k])) =$ $\mathsf{x}[i,0,k] = \mathsf{y}_1[i,0,k]\ \mathsf{op}\ \mathsf{y}_2[i,0,k]\ \ \|\|$ $\mathsf{x}[i,1,k] = \mathsf{y}_1[i,1,k]\ \mathsf{op}\ \mathsf{y}_2[i,1,k]\ \ \|\| \ldots \|\|$ $\mathsf{x}[i,J-1,k] = \mathsf{y}_1[i,J-1,k]\ \mathsf{op}\ \mathsf{y}_2[i,J-1,k]$ | *operation* |
| | $\mid \mathsf{x}[i,J,k] = \mathsf{const}$ | analogous | *constant* |
| | $\mid \mathsf{x}[i,J,k] = \mathsf{PHI}(\mathsf{x}_1[i,J,k], \mathsf{x}_2[i,J,k-1])$ | | *pseudo PHI* |
| | $\mid \mathsf{x}[i,J,k] = \mathsf{raise\_dim}(\mathsf{x}'[i], (J:J, k:K))$ | | *raise dimension*$(s)$ |
| | $\mid \mathsf{x}[i,J] = \mathsf{drop\_dim}(\mathsf{x}'[i,J,k], k)$ | | *drop dimension*$(s)$ |
| | $\mid$ for $i$ in range$(I)$ : $s$ | $\gamma(\text{for } i \text{ in range}(I) : s) =$ $\gamma(s)[0/i]\ ;\ \gamma(s)[1/i]\ ;\ \ldots\ ;\ \gamma(s)[I-1/i]$ | *loop* |

**Figure 2: MPC Source Syntax and Semantics. $\gamma$ defines the semantics of MPC source which is a linearization of MPC Source. A SIMD operation parallelizes operations across the vectorized $J$ dimension. $\|\|$ denotes parallel execution, which is standard. $\gamma$ of a for loop unrolls the loop. ; denotes sequential execution. Iterative MPC Source trivially extends to non-vectorized dimensions over the enclosing loops.**

Z3). 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 to the array PHI-node for loop $j$, thus "freeing up" statements for vectorization.

In Histogram, this removes the back edge from 4 to the PHI node at 2. Clearly there does not exist $i < i'$ that makes $i$ at 9 and $i'$ at 6 and 7 equal. The back-edge from 9 to 4 stays because for every $j < j'$ 9 and 6 and 7 access the same location in res.

Removing infeasible edges renders some array phi-nodes *target-less*. We handle target-less phi-nodes with a minor extension of Vectorization (Phase 2). First, we merge closures that update the same array. This simplifies handling of array $\phi$-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. Additional detail on the minor changes to *raise_dim* and *drop_dim* required to do expansion of arrays appear in [**?**].

Returning to the histogram example, the canonical (i.e., declared) 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, 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-d array, in this case it sets up each res3[i,j] = tmp3[i,j].

```
1  A1 = raise_dim(A, j, ((i:num_bins),(j:N)))
2  B1 = raise_dim(B, j, ((i:num_bins),(j:N)))
3  I = raise_dim(i, ((i:num_bins),(j:N)))
4  for i in range(0, num_bins):
5     res1 = PHI(res, res2^) # target−less phi−node
6     res1^ = raise_dim(res1, (j:N))
7     for j in range(0, N):
8        res2 = PHI(res1^, res3)
9        tmp1 = (A1 == I)
10       tmp2 = (res2 + B1)
11       tmp3 = MUX(tmp1, res2, tmp2)
12       res3 = Update(res2, (I,J), tmp3)
13    res2^ = drop_dim(res2)
14 res1'' = drop_dim(res1)
15 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. (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.

```
1  A1 = raise_dim(A, j, ((i:num_bins),(j:N)))
2  B1 = raise_dim(B, j, ((i:num_bins),(j:N)))
3  I1 = raise_dim(i, ((i:num_bins),(j:N)))
4
5  tmp1[I,J] = (A1[I,J] == I1[I,J])
6
7  res1^ = raise_dim(res, (j:N)) # replacing res1 with res, 1st arg
8  for j in range(0, N):
9     res2 = PHI(res1^, res3)
10    tmp2[I,j] = (res2[I,j] + B1[I,j])
11    tmp3[I,j] = MUX(tmp1[I,j], res2[I,j], tmp2[I,j])
12    res3 = Update(res2, (I,j), tmp3)
13    equiv. to res3 = res2; res3[I,j] = tmp3[I,j]
14 res2^ = drop_dim(res2)
15 res1 = drop_dim(res2^) # replacing with res2^, 2nd arg. NOOP
16 return res1
```

# 6 COMPILER BACK END

MOTION code generation requires that variables are marked as `plain` or `shared` following the type system in §??. We require that all inputs are marked as either shared or plaintext, however, we infer qualifiers for the rest of the variables. Type inference is done on MPC Source and amounts to a standard positive-negative qualifier system (`shared` is positive and `plain` is negative). Translation from MPC Source to MOTION C++ code is relatively straightforward. The extended version [?] details the taint analysis and code generation.

*Variable declarations:* The generated MOTION code begins with the declaration of all variables used in the function, including loop counters. If a variable is a vectorized array, it is initialized to a correctly-sized array of empty MOTION shares. Additionally, each plaintext variable and parameter has a shared counterpart declared. Next, all constant values which are used as part of shared expressions are initialized as a shared input from party 0. Finally, plaintext parameters are converted as shared inputs from party 0 to initialize their shared counterparts. (We note that inputs are typically read-only and the conversion may incur 2-3 redundant input gates per application; this can be fixed.)

*Code generation:* Once the function preamble is complete, the MPC Source is translated into C++ one statement at a time. The linear structure of MPC Source enables this approach to translation. E.g., MPC Source for-loops are converted to C++ for-loops which iterate the loop counter over the specified range. The psuedo-PHI node has the expected semantics as detailed in [?].

*Vectorization and SIMD operations:* Vectorization is handled with utility functions to manage accessing and updating slices of arrays. All SIMD values are stored in non-vectorized form as 1-dimensional std::vectors in row-major order. Whenever a SIMD value is used in an expression, the utility function vectorized_access() (see Listing ??) takes the multi-dimentional representation of a SIMD value, along with the size of each dimension and the requested slice's indices, and converts that slice to a MOTION SIMD value. Because MOTION supports SIMD operations using the same C++ operators as non-SIMD operations, we do not need to perform any other transformations to the expression. Therefore, once vectorized accesses are inserted the translation of an expression containing SIMD values is identical to that of expressions without SIMD values. Similarly, vectorized_assign() assigns a (potentially SIMD) value to a slice of a vectorized array. This operation cannot be done with a simple subscript as SIMD assignments will update a range of values in the underlying array representation. Listing ?? illustrates the ADD in Biometric that is vectorized along the $i$ dimension.

*Upcasting from plaintext to shared:* Currently, our compiler only supports the `Bmr` and `BooleanGMW` protocols as MOTION does not implement all operations for other protocols. MOTION does not support publicly-known constants for these protocols, so all conversions from plaintext values to shares are performed by providing the plaintext value as a shared input from party 0. To minimize conversions we create a shared copy of each plaintext variable and update that copy in lock-step with the plaintext variable. Loop counters are the one major case that trigger such update (and thus many incur runtime cost as they must be converted to a shared value on each iteration); thus, we only generate this conversion when necessary, i.e., when the counter flows to a shared computation.

Due to SSA translation and scalar expansion our generated vectorized MOTION code often includes multiple copies of arrays (typically expanded scalar values). These copies do not incur a runtime cost as the arrays simply hold *pointers* to the underlying shares, so no new shares or gates are created as a result of this copying. Cost in MPC is dominated by shares and computation on shares.

# 7 EXPERIMENTAL RESULTS

## 7.1 Experiment Setup

We tested our framework with several benchmarks. For the multiparty computation (MPC), we restricted our evaluation to 2 party computation (2PC) setting because it requires fewer computing resources. We stress that there is no such inherent restriction in our framework. We use hardware resources provided by CloudLab[?] and consider two network settings, namely Local Area Network (LAN) and Wide Area Network (WAN). In the LAN setting, we use `c6525-25g` machines for both parties. These machines are equipped with 16-core AMD 7302P 3.0GHz processors and 128GB of RAM. The connection between these machines had 10Gbps bandwidth and sub-millisecond latency. In the WAN setting we again used a `c6525-25g` machine (located in Utah, US) for the first party and a `c220g1` machine (located in Wisconsin, US) for the second. The `c220g1` machine is equipped with two Intel E5-2630 8-core 2.40GHz processors and 128GB of RAM. We measured the connection bandwidth to be 560Mbps and average round trip time (RTT) to be 38ms.

We run all experiments 5 times and report average values of various metrics. Note that standard deviation in all observations was at most 4.5% of the mean value.

## 7.2 Benchmarks

We used benchmarks from ABY and OPA [?, ?] (Biometric Matching, Histogram, PSI), HyCC [?] (Database Join, Database Variance, Inner Product, Cryptonets Max Pooling, MNIST ReLU, k-means Iteration) and Parsynt [?] (Convex Hull, Minimal Points, Count 102, Count 10, Longest 102, Max distance between symbols). We did not use Gcd [?] and Modular Exponentiation from [?] as they do not benefit from vectorization according to our model. We ran as many routines from [?] and [?] as possible. Gaussian decomposition cannot be expressed in our syntax and the compiler errors on Cryptonets Convolution at the time of writing, although we expect to fix the issue soon. [?] describes what the benchmarks do.

To compare we run *both* versions with the maximal value of $N$ for which the non-vectorized version succeeds (typically

```
sum!4[I] = ADD_SIMD(sum!3[I], p[I, j])
```

MPC Source

```
vectorized_assign(sum_4, {_MPC_PLAINTEXT_N}, {true}, {},
    vectorized_access(sum_3, {_MPC_PLAINTEXT_N}, {true}, {}) +
    vectorized_access(p, {_MPC_PLAINTEXT_N, _MPC_PLAINTEXT_D}, {true, false},
        {_MPC_PLAINTEXT_j}));
```

MOTION Code

**Table 2: MOTION Translation: Assignment to SIMD value**

N=128). We also run *vec* with larger values, typically N=4096. Again, due to space constraints [**?**] gives the exact values.

## 7.3 Results and Analysis

A detailed summary of the effects of vectorization on various benchmarks is presented in **??**. We show circuit evaluation times in **??**. In terms of observed benefits from vectorization, we divide benchmarks into 3 categories: 1) *High:* these include Convex Hull, Cryptonets Max Pooling, Minimal Points and Private Set Intersection. 2) *Medium:* these include Biometric Matching, DB Variance, Histogram, Inner Product, k-means Iteration and MNIST ReLU. These benchmarks have non-parallelizable phases e.g. the summing phase of Inner Product. Still, most computation is parallelizable and it results in speedup from 5x to 25x in BMR, and 2x to 25x in GMW protocol. 3) *Low:* these include the Database Join and the regular expression benchmarks (Count 102, Count 10, Longest 102 and Max distance between symbols). There is less parallelizable computation in these programs, thus the speedup is lower. We see a speedup from 1.1x to 2x in BMR. In GMW, DB Join, Count 102 and Count 10s see speedup from 1.1x to 1.3x. However, Longest 102 and Max distance between symbols suffer a slowdown of 0.5x. There is opportunity for vectorization in these benchmarks according to our analytical model, particularly, there is a large EQ that is vectorized, although a large portion of the loop cannot be vectorized. We observed that transformation to vectorized code increased multiplicative depth and, the negative effect of increased depth is more noticeable in a round-based protocol like GMW. The cause of the increase is not clear — we conjecture that MOTION performs optimization over the non-vectorized loop body that decreases depth; also, EQ is relatively inexpensive in Boolean GMW and BMR compared to ADD and MUL, which also de-emphasizes the benefit of vectorization. We propose a simple heuristic: if the transformation increases circuit depth beyond some threshold (e.g. more than 10% of the original circuit), we can reject the transformation. Nevertheless, we show these benchmarks here for the sake of completeness and to highlight that vectorization does not always reduce run time. Note that in some settings it may still be desirable to vectorize e.g. in data constrained environments as communication as well as all other metrics, is reduced for all benchmarks. (The extended version presents plots of for all metrics [**?**].)

We look closely at the Biometric Matching benchmark: circuit evaluation in **??**, communication size in [**?**] and circuit generation time in [**?**]. For input size beyond N=128 the memory usage exceeds available memory and prevents circuit generation. Consequently, non-vectorized bars are missing beyond this threshold. Notice that vectorization improves all metrics. Comparing performance improvement between BMR and GMW, we see more speedup for BMR (23x vs 10x), GMW gets more communication size reduction (10x vs 2.5x) and circuit generation sees a speedup of 35x and 45x for BMR and GMW respectively.

Since our vectorization framework is network agnostic, it produces the same circuit for both LAN and WAN. This means that the number of gates and communication size remain the same. Moreover, time for circuit generation, which is a local operation, also remains unchanged. Setup and Online times, however, increase due to lower bandwidth and higher latency of the WAN. Indeed, this is what we observe in **??**.

## 7.4 Comparison with MOTION-native Inner Product and Discussion

Finally, we compared our automatically generated routine for Inner Product with the manually SIMD-ified MOTION-native routine in the distribution. We were surprised that we were an order of magnitude slower in Boolean GMW as our circuit ran a significantly larger number of communication rounds. Upon investigation, it turned out that the vectorized multiplications were essentially the same, however, our addition loop incurred significant cost (ADD is non-local and expensive in Boolean GMW). The MOTION-native loop ran `result += mult_unsimdified[i];` while our loop ran auto generated `result[i] = result_prev[i] + mult_unsimdified[i];` (as scalar expansion is an artifact of our vectorization). We rewrote the accumulation manually and that led to the comparable speedup!

MOTION's compiler performs analysis that informs circuit generation and the example illustrates the power of the analysis. In the above example, MOTION does the standard divide-and-conquer accumulation. We conjectured that poor performance of our loop was due to limitations of our implementation. Recall that Phase 3 of Vectorization in §**??**, which we have not implemented yet, gets rid of redundant dimensions and generates `if (i!=0) { result_prev = result; } result = result_prev + mult_unsimdified[i]`. And while it is unrealistic to expect that static analysis will detect the associative accumulation in the scalar expansion code, one might expect that it will in the above simpler code. However, because AST analysis is difficult, it does not appear to detect the accumulation.

**Table 3: Vectorized vs Non-Vectorized Comparison, times in seconds (in LAN setting where applicable), Communication in MiB, Numbers in 1000s, values rounded to nearest integer, benchmark names ending in *V* are vectorized.**

| | GMW | | | | | | BMR | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | Online | Setup | # Gates | Circ Gen | # Msgs | Comm. | Online | Setup | # Gates | Circ Gen | # Msgs | Comm. |
| Biometric Matching | 146 | 16 | 1,784 | 119 | 1,413 | 140 | 89 | 263 | 1,595 | 139 | 2,716 | 312 |
| Biometric Matching (V) | 12 | 4 | 34 | 2 | 28 | 14 | 2 | 13 | 30 | 4 | 61 | 130 |
| Convex Hull | 48 | 6 | 551 | 40 | 516 | 51 | 28 | 72 | 494 | 39 | 695 | 80 |
| Convex Hull (V) | 0 | 1 | 2 | 0 | 1 | 4 | 0 | 2 | 1 | 1 | 2 | 32 |
| Count 102 | 79 | 6 | 418 | 35 | 525 | 52 | 15 | 62 | 269 | 33 | 785 | 92 |
| Count 102 (V) | 71 | 5 | 316 | 24 | 332 | 34 | 11 | 30 | 167 | 16 | 304 | 59 |
| Count 10s | 79 | 6 | 419 | 35 | 525 | 52 | 14 | 62 | 270 | 33 | 785 | 92 |
| Count 10s (V) | 71 | 4 | 316 | 24 | 332 | 34 | 11 | 29 | 167 | 16 | 304 | 59 |
| Cryptonets (Max Pooling) | 50 | 11 | 688 | 46 | 554 | 55 | 36 | 89 | 608 | 51 | 898 | 110 |
| Cryptonets (Max Pooling) (V) | 1 | 1 | 7 | 1 | 2 | 5 | 2 | 4 | 7 | 2 | 12 | 49 |
| Database Join | 70 | 8 | 433 | 48 | 790 | 80 | 19 | 229 | 458 | 119 | 3,518 | 427 |
| Database Join (V) | 54 | 6 | 320 | 35 | 575 | 61 | 16 | 112 | 320 | 57 | 1,457 | 285 |
| Database Variance | 166 | 18 | 2,009 | 135 | 1,639 | 163 | 95 | 269 | 1,708 | 145 | 2,795 | 320 |
| Database Variance (V) | 37 | 6 | 321 | 24 | 334 | 43 | 10 | 30 | 170 | 13 | 178 | 141 |
| Histogram | 94 | 10 | 862 | 68 | 979 | 97 | 27 | 94 | 491 | 51 | 1,132 | 135 |
| Histogram (V) | 33 | 5 | 166 | 16 | 164 | 23 | 7 | 17 | 92 | 13 | 154 | 68 |
| Inner Product | 127 | 15 | 1,675 | 108 | 1,308 | 130 | 83 | 250 | 1,526 | 134 | 2,623 | 301 |
| Inner Product (V) | 16 | 5 | 158 | 12 | 165 | 25 | 6 | 18 | 83 | 7 | 86 | 127 |
| k-means | 108 | 12 | 1,333 | 88 | 1,090 | 108 | 63 | 185 | 1,141 | 99 | 1,958 | 225 |
| k-means (V) | 6 | 3 | 47 | 4 | 43 | 12 | 2 | 11 | 32 | 4 | 54 | 95 |
| Longest 102 | 93 | 7 | 650 | 52 | 713 | 71 | 26 | 93 | 475 | 49 | 1,091 | 128 |
| Longest 102 (V) | 169 | 6 | 544 | 41 | 519 | 53 | 25 | 60 | 369 | 33 | 605 | 95 |
| Max. Dist. b/w Symbols | 71 | 8 | 572 | 43 | 576 | 57 | 24 | 69 | 397 | 38 | 748 | 89 |
| Max. Dist. b/w Symbols (V) | 166 | 7 | 538 | 39 | 512 | 51 | 24 | 57 | 363 | 32 | 589 | 78 |
| Minimal Points | 35 | 5 | 458 | 31 | 369 | 37 | 24 | 46 | 401 | 26 | 347 | 40 |
| Minimal Points (V) | 0 | 1 | 1 | 0 | 1 | 3 | 0 | 1 | 1 | 0 | 1 | 16 |
| MNIST ReLU | 132 | 31 | 1,843 | 126 | 1,483 | 152 | 98 | 247 | 1,630 | 135 | 2,401 | 298 |
| MNIST ReLU (V) | 3 | 3 | 25 | 3 | 9 | 17 | 5 | 11 | 25 | 5 | 33 | 136 |
| Private Set Intersection | 95 | 9 | 558 | 59 | 1,049 | 104 | 22 | 186 | 591 | 96 | 2,639 | 302 |
| Private Set Intersection (V) | 1 | 2 | 1 | 2 | 1 | 8 | 1 | 8 | 2 | 4 | 2 | 122 |

We conjecture that MPC Source, a straight-forward representation suitable for static analysis, will not only enable detection of general associative loops, but also allow for program synthesis to increase opportunities for divide-and-conquer parallelization [**?**]; we leave this and other optimizations as future work.

HyCC [**?**] is a mixing compiler that focuses on mixed protocols, while we run vectorization within a single protocol. The paper only provides data for Boolean and Yao for a version of Biometric matching on $N = 1000$ (as its focus is mixing). We estimate we are about an order of magnitude slower, which is likely due to the same issue — the computation of *min* can be optimized.

## 8 RELATED WORK

*MPC languages and compilers.* Languages and compilers for secure computation have seen significant 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 [**?**]. We believe

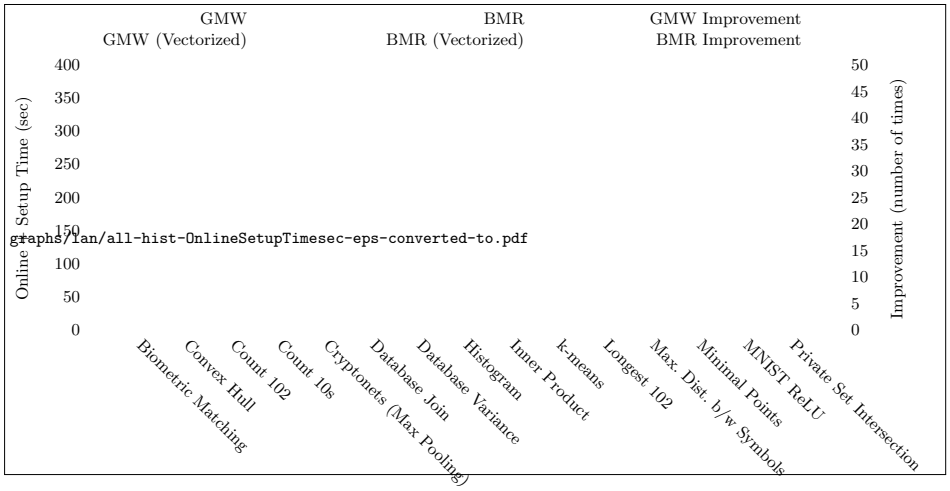



**Figure 3: Circuit Evaluation Time (Setup + Online) of Benchmarks**

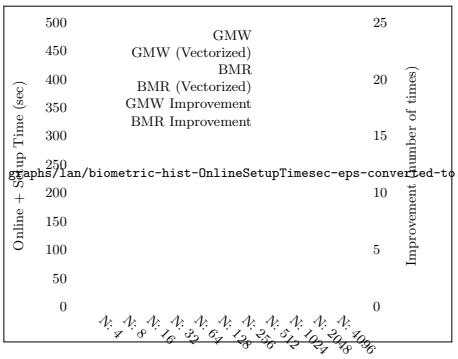



**Figure 4: Biometric Matching Circuit Evaluation Time**

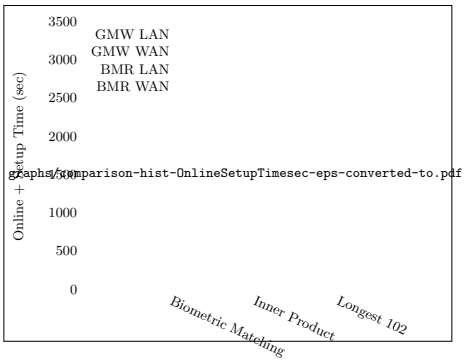



**Figure 5: LAN vs. WAN: Circuit Evaluation Time**

that our work is unique as as it falls between the high-level language and lower-level circuit protocols.

HyCC [**?**] is a compiler from C Source into ABY circuits. It does source-to-source compilation with the key goal to decompose the program into modules and then assign protocols to modules. In contrast, we focus on MPC Source-level optimizations, specifically vectorization, although we envision future optimizations as well. We formalize MPC Source and reasoning about transformations, which we conjecture is more tractable than reasoning over the higher-level AST. On the other hand, HyCC does inter-procedural optimizations, while our analysis is intra-procedural. We will explore context-sensitive inter-procedural analysis over MPC Source in future work. HyCC, similarly to Buscher [**?**] uses an of-the-shelf source-to-source polyhedral compiler [2] to perform vectorization at the level of source code. The disadvantage of using an of-the-shelf source-to-source compiler is that it solves a more general problem than what MPC presents and may forgo opportunities for optimization — concretely, it is well-known that vectorization and polyhedral compilation do not work well with conditionals [**?**, **?**]. In contrast, we consider vectorization at the level of MPC Source which linearizes conditionals; we are able to handle programs with interleaved if-statements and for-loops.

*Classical HPC compilers.* Automatic vectorization is a longstanding problem in high-performance computing and there are thousands of works reflecting many 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 as well as selection (similar to MUX).

We believe that vectorization over linear MPC Source warrants a new look, while drawing from results in HPC. Polyhedral parallelization [**?**] considers a higher-level source

[2] We believe HyCC uses Par4All (https://github.com/Par4All/par4all), however, does not appear to be included with the publicly available distribution of HyCC.

(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. We consider different array representation, notion of dependence, and reasoning about dependence, which is more suitable for MPC Source.

# 9 CONCLUSION AND FUTURE WORK

We presented a formalization of the MPC Source intermediate language followed by a specific back-end optimization at the level of MPC Source: novel SIMD-vectorization. We demonstrated that vectorization has significant impact on performance. We are excited about many opportunities for future work — integration with protocol mixing, divide-and-conquer reasoning and parallelization, as well as inter-procedural context-sensitive analysis for MPC Source.