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 the idea of 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 speedup in circuit generation time and running time, as well as significant reduction in communication and number of gates over the corresponding iterative schedule.

1 INTRODUCTION

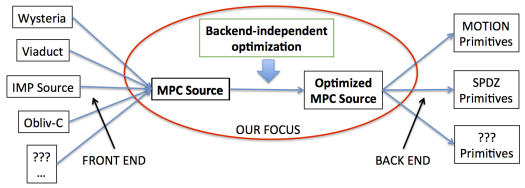
Multi-party computation (MPC) allows N parties $p_1, \ldots, p_N$ to perform a computation on their private inputs securely. Informally, security means that the secure computation protocol computes the correct output (correctness) and it does not leak any information about the individual party inputs,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

Earlier compilers did both back-end and front-end translation 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 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, other) 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. MPC Source facilitates cost modeling. 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 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 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 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 do not provide the link only to preserve anonymity, however, we will gladly make it available upon request from reviewers. The Github setup generates graphs and intermediate code for each program along each compiler phase and runs MOTION on small inputs to generate tables of data (the experiments in the paper 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. (We detail the restrictions in the following sections.) 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. The code iterates over the entries in the database and computes the Euclidean distance of the current entry S[i] and C (its square actually). 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 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 β for a local MPC operation (e.g., ADD in Arithmetic sharing) and cost α for a remote MPC operation (e.g., MUX, CMP, etc.). Assuming that ADD is α and SUM, CMP and MUX are β, 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 $N * D$ subtraction operations (line 9 in (b)) in a single SIMD instruction; similarly we can compute all multiplication operations (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

```
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 13-14 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 ϕ-nodes at the if-then-else (lines 13-15) turn into MUX operations, while the ϕ-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.

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 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 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 $\alpha_{var} \approx 0$ and orders of magnitude improvement (e.g., 12x in online time in GMW). Additionally, the un-vectorized version runs out of memory for $N = 256$, while the vectorized one runs with the standard maximal input size $N = 4,096$.

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 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, 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 A to party B 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 $i' < j'$.

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

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

where L_α is the number of α-instructions and L_β is the number of β ones. (We used this formula to compute the cost of the unrolled MPC Source program in §??.) The cost of schedule P is more interesting:

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

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 $|P_i|$ α-instructions, or $cost(P_i) = \beta_{fix} + |P_i|\beta_{var}$ if it stores β-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 is 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 α-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

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

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

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. 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. We start the section with a brief description of the top-level syntax and semantic restrictions in §**??**. We proceed to describe the front end of the compiler: §**??** describes translation from IMP Source to SSA, and §**??** describes translation form SSA into MPC Source.

We describe analysis on MPC Source and backend-independent vectorization in §**??** and translation into MOTION code in §**??**.

4.1 Syntax and Semantic Restrictions

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

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

$\tau ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{list[int]} \mid \mathtt{list[bool]}$	*base types*
$q ::= \mathtt{shared} \mid \mathtt{plain}$	*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 $\mathtt{plain} <: \mathtt{shared}$, 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 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 as specified by the rule:

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

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.)

(4) The final restriction involves MUX as expressed by the rule:

$$\frac{\Gamma \vdash e_1 : \langle q_1\ \mathtt{bool} \rangle \quad \Gamma \vdash e_2 : \langle q_2\ \tau \rangle \quad \Gamma \vdash e_2 : \langle q_2\ \tau \rangle \quad \tau \in \{\mathtt{int}, \mathtt{bool}\}}{\Gamma \vdash \mathtt{MUX(}e_1\mathtt{,}e_2\mathtt{,}e_3\mathtt{)} : \langle q_1 \vee q_2 \vee q_3\ \tau \rangle}\ (\textsc{MUX})$$

The arguments of MUX are restricted to base types. (Again, this is just a restriction of our current implementation that will be lifted.) This causes an inconvenience as we could not write

```
1 if e: A[i] = val
```

Instead we had to write

```
1 if not(e): val = A[i]
2 A[i] = val
```

We note that while the programmer writes annotations at the level of IMP Source (as in Listing **??**(a)), the annotations propagate through the transformations; annotations are inferred and checked with a taint analysis at the level of MPC Source. Tthe programmer annotates only inputs.

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

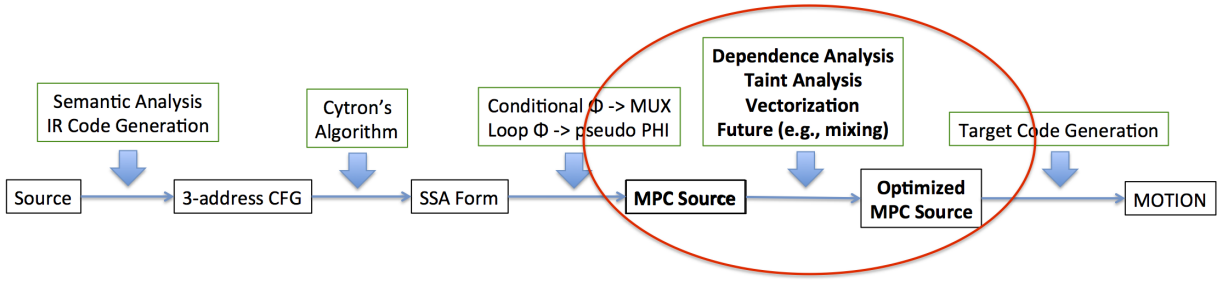


Figure 1: Compiler Framework.

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 follows:

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 §??. This step outputs an instance of the `restricted_ast.Function` class, which represents our restricted subset of the Python AST.

3-address CFG conversion: 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 (as in §??). Rules for processing each kind of statement are given below:

(1) **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.

(2) **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.

(3) **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, e.g., 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.

SSA conversion: Convert the 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 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 = ϕ(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.

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 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 ϕ-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 ϕ-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 these to the main MPC source.

Now, the remaining ϕ-nodes in MPC source are the loop header nodes. We call these nodes *pseudo* ϕ-nodes and we write PHI in MPC Source. 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.

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 an ad-hoc way, e.g., [**?**, **?**, **?**]).

§**??** describes our dependence analysis and §**??** describes scalar expansion, which lifts scalars (and arrays) to the corresponding loop dimensionality to create opportunities for vectorization. §**??** describes our core vectorization algorithm and §**??** argues correctness of the vectorization transformation. §**??** extends vectorization with array writes.

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 **??** 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., 15 to 4 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. E.g., 1 to 4 in Biometric forms is an outer-to-inner edge.
- 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., the def-use from 8 to 12 gives rise to 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 ϕ-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 $x' = x$ immediately after the i, j, k loop. Then we replace the use of x at Y with 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.

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 the scalar expansion 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.

Arrays. Conceptually, we treat all variables as arrays. There are three kinds of arrays.

- Scalars: These are scalar variables we expand into arrays for the purposes of vectorization. For those, all writes are canonical writes and all reads are canonical reads. We will *raise dimension* when a scalar gives rise to an outer-to-inner dependence edge (e.g., sum!2 in line 6 of the MPC source code will be raised to a 1-dimensional array since sum!2 is used in the inner j-loop). We will *drop dimension* when a scalar gives rise to an inner-to-outer dependence edge (e.g., sum!2 for which the lifted inner loop computes D values, but the outer loop only needs the last one.)
- Read-only input arrays: There are no writes, while we may have non-canonical reads, $f(i, j, k)$. Vectorization adds raise dimension operations at the beginning of the function to lift these arrays to the dimensionality of the loop where they are used, possibly *reshaping* the 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.
- Read-write output arrays: Writes are canonical (by restriction) but reads can be non-canonical. Dependences limit 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 to lower dimension. 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.

Raise dimension. The *raise_dim* function expands a scalar (or array). There are two conceptual versions of *raise_dim*. One reshapes an arbitrary access into a canonical read access in the corresponding loop. It takes the original array, the access pattern function $f(i,j,k)$ in loop nest i,j,k and the loop bounds $((i:I),(j:J),(k:K))$:

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

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

$$\mathsf{A}'[i,j,k] = \mathsf{A}[f(i,j,k)]$$

The end result is that uses of $\mathsf{A}[f(i,j,k)]$ in loop nest i,j,k are replaced with canonical read-accesses to $\mathsf{A}'[i,j,k]$ that can be vectorized. In the running Biometric example, $\mathsf{C}' = raise\_dim(\mathsf{C}, j, (i:N, j:D))$ lifts the 1-dimensional array C into a 2-dimensional array. The i,j loop now accesses C' in the canonical way, $\mathsf{C}'[i,j]$. Similarly, $\mathsf{S}' = raise\_dim(\mathsf{S}, i{*}D{+}j, (i:N, j:D))$ 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* lifts a lower-dimension array into a higher-dimension for access in a nested loop. It is necessary when processing outer-to-inner dependences. Here A is an i-array and raise dimension adds two additional dimensions:

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

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

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

Drop dimension. 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:

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

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:

$$\mathsf{A}'[i] = A[i, J-1, K-1]$$

5.3 Basic Vectorization

We present our algorithm followed by an example on Biometric.

5.3.1 Algorithm

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*. 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. 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 cleans up local arrays of references (this is an optional phase and our current implementation does not include it) and Phase 4 explicitly turns operations into MOTION SIMD operations.

{ 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 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 $\mathsf{y}'_n[i,j,k] = raise\_dim(\mathsf{y}_n)$ 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
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) $x' = drop\_dim(x, i)$. }
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

5.3.2 Example: Biometric

We now demonstrate the workings of the basic algorithm on our running example. Recall the MPC Source for Biometric:

```
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)
```

Phase 1 of Vectorization Algorithm. 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 statements (e.g., min_sum!3 = sum!3) are implicitly vectorized. The example illustrates the two different versions of *raise_dim*. E.g., raise_dim(C, j, (i:N,j:D)) reshapes the read-only input array. drop_dim(sum!3) removes the j dimension of sum!3.

```
1  min_sum!1 = MAX_INT
2  min_sum!1^ = raise_dim(min_sum!1, (i:N))
3  min_idx!1 = 0
4  min_idx!1^ = raise_dim(min_idx!1, (i:N))
5  S^ = raise_dim(S, ((i * D) + j), (i:N,j:D))
6  C^ = raise_dim(C, j, (i:N,j:D))
7  for i in range(0, N):
8      min_sum!2 = PHI(min_sum!1^, min_sum!4)
9      min_idx!2 = PHI(min_idx!1^, min_idx!4)
10     sum!2 = 0 // Will lift, when hoisted
11     sum!2^ = raise_dim(sum!2, (j:D))
12     for j in range(0, D):
13         sum!3 = PHI(sum!2^, sum!4)
14         d = SUB(S^,C^)
15         p = MUL(d,d)
16         sum!4 = ADD(sum!3,p)
17     sum!3^ = drop_dim(sum!3)
18     t = CMP(sum!3^,min_sum!2)
19     min_sum!3 = sum!3^
20     min_idx!3 = i // Same−level, will lift when hoisted
21     min_sum!4 = MUX(t, min_sum!3, min_sum!2)
22     min_idx!4 = MUX(t, min_idx!3, min_idx!2)
23 min_sum!2^ = drop_dim(min_sum!2)
24 min_idx!2^ = drop_dim(min_idx!2)
25 return (min_sum!2^, min_idx!2^)
```

Phase 2 of Vectorization Algorithm. This phase analyzes statements from the innermost loop to the outermost. The key point is to discover loop-carried dependencies and reintroduce loops whenever dependencies make this necessary.

Starting at the inner phi-node sum!3 = PHI(...), the algorithm first computes its closure. The closure amounts to the phi-node itself and the addition node sum!4 = ADD(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 vectorization on j. Note that the SUB and MUL computations remain outside of the loop as they do not depend on PHI-nodes that are part of cycles. The dependences are from p[I,J] = MUL(d[I,J],d[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 the inner loop code becomes:

```
min_sum!1 = MAX_INT
min_sum!1^ = raise_dim(min_sum!1, (i:N!0))
min_idx!1 = 0
min_idx!1^ = raise_dim(min_idx!1, (i:N))
S^ = raise_dim(S, ((i * D) + j), (i:N,j:D))
C^ = raise_dim(C, j, (i:N,j:D))
for i in range(0, N):
  min_sum!2[I] = PHI(min_sum!1^[I], min_sum!4[I])
  min_idx!2[I] = PHI(min_idx!1^[I], min_idx!4[I])
  sum!2 = [0,..,0]
  sum!2^ = raise_dim(sum!2, (j:D))
  d[I,J] = SUB(S^[I,J],C^[I,J])
  p[I,J] = MUL(d[I,J],d[I,J])
  for j in range(0, D):
    sum!3[I,j] = PHI(sum!2^[I,j], sum!4[I,j−1])
    sum!4[I,j] = ADD(sum!3[I,j],p[I,j])
  sum!3^ = drop_dim(sum!3)
  t[I] = CMP(sum!3^[I],min_sum!2[I])
  min_sum!3 = sum!3^
  min_idx!3 = i
  min_sum!4[I] = MUX(t[I], min_sum!3[I], min_sum!2[I])
  min_idx!4[I] = MUX(t[I], min_idx!3[I], min_idx!2[I])
min_sum!2^ = drop_dim(min_sum!2)
min_idx!2^ = drop_dim(min_idx!2)
return (min_sum!2^, min_idx!2^)
```

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

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

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

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

sum!3^ = drop_dim(sum!3)
min_idx!3 = [0,1,2,...N−1] // i.e., min_idx!3 = [i, (i:N)]
min_sum!3 = sum!3^

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

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

min_sum!2^ = drop_dim(min_sum!2)
min_idx!2^ = drop_dim(min_idx!2)
return (min_sum!2^, min_idx!2^)
```

Phase 3 of Vectorization Algorithm. 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 = MAX_INT
min_idx!1 = 0
S^ = raise_dim(S, ((i * D) + j), (i:N,j:D))
C^ = raise_dim(C, j, (i:N,j:D))

sum!2 = [0,..,0]
d[I,J] = SUB(S^[I,J],C^[I,J])
p[I,J] = MUL(d[I,J],d[I,J])

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

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

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

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

// drop_dim becomes redundant
return (min_sum!2, min_idx!2)
```

Phase 4 of Basic Vectorization. This phase adds SIMD operations:

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

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

```

```
for j in range(0, D):
  // 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[I,j])

min_idx!3 = [0,1,...N−1]

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

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

return (min_sum!2, min_idx!2)
```

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. 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. **??** 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 $a_0 \leq a_1$.*

Theorem 1 states that a def-use edge in MPC Source a_0 gives rise to a set of def-use edges in $\gamma(a_0)$ and a "same" edge in a_1 gives rise to a set of def-use edges in $\gamma(a_1)$ and the former set is included in the latter one. This means that the computation of $\gamma(a_0)$ is carried out by $\gamma(a_1)$ as well, computing the same result. The theorem allows that a_1 and $\gamma(a_1)$ have larger state that a_0 and $\gamma(a_0)$, i.e., doing additional computation into additional memory locations. Theorem 2 establishes the correctness of vectorization — clearly, vectorization preserves all statements and def-use edges in the original MPC Source, and therefore the linearization computes the same result as the original MPC Source. (In this case we have an equivalence, $a_0 = a_1$ according to our partial order.)

s	$::= s_1; s_2$	$\gamma(s) = \gamma(s_1)\ ;\ \gamma(s_2)$	*sequence*
	$\mid \mathsf{x}[i,J,k] = \mathsf{op\_SIMD}(\mathsf{y}_1[i,J,k], \mathsf{y}_2[i,J,k])$	$\gamma(\mathsf{x}[i,J,k] = \mathsf{op\_SIMD}(\mathsf{y}_1[i,J,k], \mathsf{y}_2[i,J,k])) =$ $\mathsf{x}[i,0,k] = \mathsf{y}_1[i,0,k]\ \mathsf{op}\ \mathsf{y}_2[i,0,k]\ \ \|\|$ $\mathsf{x}[i,1,k] = \mathsf{y}_1[i,1,k]\ \mathsf{op}\ \mathsf{y}_2[i,1,k]\ \ \|\| \ ... \ \|\|$ $\mathsf{x}[i,J-1,k] = \mathsf{y}_1[i,J-1,k]\ \mathsf{op}\ \mathsf{y}_2[i,J-1,k]$	*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	$\gamma(\text{for } i \text{ in range}(I) : s) =$ $\gamma(s)[0/i]\ ;\ \gamma(s)[1/i]\ ;\ ...\ ;\ \gamma(s)[I-1/i]$	*loop*

Figure 2: MPC Source Syntax and Semantics. γ 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. γ of a for loop unrolls the loop. ; denotes sequential execution. Iterative MPC Source trivially extends to non-vectorized dimensions over the enclosing loops.

5.5 Extension with Array Writes

5.5.1 Removal of Infeasible Edges

Array writes limit vectorization as they sometimes introduce infeasible loop-carried dependencies. Consider the classical example from [**?**]:

```
1 for i in range(N):
2   A[i] = B[i] + 10;
3   B[i] = A[i] * D[i−1];
4   C[i] = A[i] * D[i−1];
5   D[i] = B[i] * C[i];
```

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

```
1 for i in range(N):
2   A_1 = PHI(A_0,A_2)
3   B_1 = PHI(B_0,B_2)
4   C_1 = PHI(C_0,C_2)
5   D_1 = PHI(D_0,D_2)
6   A_2 = update(A_1, i, B_1[i] + 10);
7   B_2 = update(B_1, i, A_2[i] * D_1[i−1]);
8   C_2 = update(C_1, i, A_2[i] * D_1[i−1]);
9   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 7 to the read of ... = B_1[i] at 6.

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

Consider the earlier example. There is a single loop, i. Clearly, there is no pair $\underline{i}$ and $\underline{i}'$, where $\underline{i} < \underline{i}'$ that make $\underline{i} = \underline{i}'$ due to the def-use pairs of A 6-7 and 6-8. Therefore, we remove the back edge from 6 to the phi-node 2. Analogously, we remove the back edges from 7 to 3 and from 8 to 4. However, there are many values $\underline{i} < \underline{i}'$ that make $\underline{i} = \underline{i}' - 1$ and the back edge from 9 to 5 remains (def-use pairs for D). As a result of removing these spurious edges, Vectorization will find that statement 6 is vectorizable. Statements 7, 8 and 9 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 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 $X \to Y$ 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 $X \to Y$ for an array A.

(1) same-level $X \to Y$. Do nothing, propagate the array, which happens to be of the right dimension.
(2) inner-to-outer $X \to Y$ 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 $X \to Y$ 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" $X \to Y$. 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 A$[f(i,j,k)]$ different than a canonical read A$[i,j,k]$, then we need to add a reshape operation as all arrays are 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 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 Vectorization the Aiken's array write example will be (roughly) as follows:

```
for i in range(N):
  A_1 = PHI(A_0,A_2)
  B_1 = PHI(B_0,B_2)
  C_1 = PHI(C_0,C_2)
  D_1 = PHI(D_0,D_2)
  A_2 = update(A_1, I, B_1[I] + 10);
  B_2 = update(B_1, I, A_2[I] * D_1[I−1]);
  C_2 = update(C_1, I, A_2[I] * D_1[I−1]);
  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 5; $cl = \{5,7,8,9\}$ while 6 is vectorizable. Recall that 2,3, and 4 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.

```
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 in range(N): // MOTION loop
  B_1 = PHI(B_0,B_2)
  C_1 = PHI(C_0,C_2)
  D_1 = PHI(D_0,D_2)
  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];
  C_2 = update(C_1, i, A_2[i] * D_1[i−1]);
  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, 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 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. §**??** briefly describes the taint analysis and §**??** describes MOTION code generation.

6.1 Taint Analysis

The taint analysis works on MPC Source, which lacks if-then-else control flow. This significantly simplifies treatment as there is no need to handle conditionals and implicit flow. Specifically, the compiler uses the following rules, which are standard in positive-negative qualifier systems (here `shared` is the positive qualifier and `plain` is the negative one):

(1) Loop counters are always `plain`.
(2) If any variable on the right-hand side rhs of an assignment is shared, then the assigned variable lhs is `shared` following subtyping rule rhs <: lhs.
(3) Any variables that cannot be determined as shared via the above rules are `plain`.

In the below snippet sum!2 and sum!3 form a dependency cycle and there is no `shared` value that flows to either one. They are inferred as plaintext.

```
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]
```

When converting to MOTION code, any plaintext value used in the right-hand side of a shared assignment is converted to a shared value for that expression.

6.2 From (Optimized) MPC Source to MOTION

MOTION supports FOR loops and SIMD operations, so translation from MPC source to MOTION C++ code is relatively straightforward.

Variable declarations: Our generated C++ uses the following variable-naming scheme: shared variables are named the same as in the MPC Source with the ! replaced with an underscore (e.g. sum!2 would be translated to sum_2). Plaintext variables follow the same naming convention as shared variables but are prefixed with _MPC_PLAINTEXT_. The shared representation of constants are named _MPC_CONSTANT_ followed by the literal constant (e.g. the shared constant 0 would be named _MPC_CONSTANT_0).

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 used as shared inputs from party 0 to initialize their shared counterparts.

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. If there is no vectorization present in a statement, translation to C++ is straightforward: outside of MUX statements and array updates, non-vectorized assignments, expressions, and returns directly translate into their C++ equivalents. Non-vectorized MUX statements are converted to MOTION's MUX member function on the condition variable. Array updates are translated into two C++ assignments: one to update the value in the original array and one to assign the new array as shown in Listing **??**.

MPC FOR loops are converted to C++ FOR loops which iterate the loop counter over the specified range. Pseudo PHI nodes are broken into two components: the "FALSE" branch which assigns the initial value of the PHI node and the "TRUE" branch which assigns the PHI node's back-edge. The assignment of the "FALSE" branch occurs right before the PHI node's enclosing loop. As these assignments may rely on the loop counter, the loop counter is initialized before these statements. Inside of the PHI node's enclosing loop, a C++ `if` statement is inserted to only assign the true branch of the PHI node after the first iteration. Listing **??** illustrates this translation.

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() 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, the vectorized_assign() function 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.

Updating SIMD arrays is also implemented differently from updating non-vectorized arrays. Instead of separating the array update from the assignment of the new array, these steps are combined with the vectorized_update() utility function. This function operates identically to vectorized_assign(), however it additionally returns the array after the assignment occurs. This value is then used for the assignment to the new variable. Listing **??** illustrates vectorized_assign() and vectorized_update() on the Biometric example.

Reshaping and raising dimensions: Raising the dimensions of a scalar or array uses the lift() utility function which takes a lambda for the raised expression and the dimensions of the output. This function is also used for the scalar expansion of values which have been lifted out of FOR loops as described in §**??**. This function evaluates the expression for each permutation of indices along the dimensions and returns the resulting array in row-major order. The lambda accepts an array of integers representing the index along each of the dimensions being raised, and the translation of the expression which is being raised replaces each of the dimension index variables with the relevant subscript of this array. There is also a special case of the lift() function which occurs when we are raising an array. In this case, instead of concatenating the array for each index, we extend the array along all dimensions being raised which are not present in the array already. For example, when raising an array with dimensions $N \times M$ to an array with dimensions $N \times M \times D$, the input array will simply be extended along the D dimension: $\mathsf{A}'[n, m, d] = \mathsf{A}[n, m]$ for every d. If the input array is already correctly sized it will be returned as-is.

Dropping dimensions use the drop_dim() and drop_dim_monoreturn() utility functions. They function identically but the latter returns a scalar for the case when the final dimension of an array is dropped. These functions take the non-vectorized representation of an array, along with the dimensions of that array, and return the array with the final dimension dropped.

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. Due to this limitation, our translation to MOTION code attempts to minimize the number of conversions from a plaintext value. This is accomplished by creating a shared copy of each plaintext variable and updating that copy in lock-step with the plaintext variable. Since variables are often initialized to a common constant value (e.g. 0), this approach decreases the number of input gates by only creating a shared input for each initialization constant. Loop counters must still be converted to a shared value on each iteration that they are used, however we only generate this conversion when necessary, i.e., when the counter flows to a shared computation. This is to prevent unnecessary increase in the number of input gates when loop counters are only used as plaintext.

Due to the SSA translation phase as well as the conversions to and from SIMD values which our utility functions perform, our generated vectorized MOTION code often includes multiple copies of arrays and scalar values. These copies do not incur a runtime cost as the arrays simply hold *pointers* to the

```
A[i] = val
```
IMP Source

```
A!2 = update(A!1, i, val)
```
MPC Source

```
A_1[i] = val;
A_2 = A_1;
```
MOTION Code

Table 2: MOTION Translation: Array Updates

```
for i in range(N):
    tmp = PHI(arr[i], val!0)
    ...
```
MPC Source

```
_MPC_PLAINTEXT_i = 0;
tmp = arr[_MPC_PLAINTEXT_i];
for (; _MPC_PLAINTEXT_i < _MPC_PLAINTEXT_N; _MPC_PLAINTEXT_i++) {
    if (_MPC_PLAINTEXT_i != 0) {
        tmp = val_0;
    }
    ...
}
```
MOTION Code

Table 3: MOTION Translation: FOR loop with Phi nodes

```
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 4: MOTION Translation: Assignment to SIMD value

```
raise_dim(i + j, (i:N, j:M))
```
MPC Source

```
lift(std::function([&](const std::vector<std::uint32_t > &idxs) {return idxs[0] + idxs[1];}),
        {_MPC_PLAINTEXT_N, _MPX_PLAINTEXT_M})
```
MOTION Code

Table 5: MOTION Translation: Raising dimensions

underlying shares, so no new shares or gates are created as a result of this copying. Cost in MPC programs 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. This setting reflects typical LAN use case considering that 10Gbps LAN is increasingly common in business networks and is now available even in some home networks. 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 between these machines to be 560Mbps and average round trip time (RTT) to be 38ms. At the time of this writing, all major internet providers in the US offer 1Gbps connections to home consumers, therefore this setting reasonably reflects the typical WAN use case.

We run all experiments 5 times and report average values of various metrics. Note that standard deviation — shown as error bar on top of the bars in the graphs — in all observations was at most 4.5% of the mean value, therefore the accuracy of the results is not effected by the relatively fewer runs.

7.2 Benchmarks

In the following, we say *both* for an experiment in which we execute both non-vectorized and vectorized protocols and *vec* for the vectorized only experiment. In a preliminary

experiment we ran `N=2, N=4`, all the way to `N=4096`; the non-vectorized version ran out of memory at the value *both* and we fixed this value for these experiments. The vectorized one completed for `N=4096` for nearly all benchmarks (numbers are not shown for space reasons; however, they times are largely consistent, e.g., if *both* is $N = 2^k$ and it completes in X seconds for vectorized, then $N = 2^{12}$ completes in $(12 - k)X$ seconds). We used the following benchmarks in our evaluation:

(1) *Biometric Matching:* Server has a database S of `N` records, each record's dimension is `D`. Client submits a query C, client and server compute the closest record to C in an MPC. We use `N=128` for *both* and `N=4096` for *vec*. `D` is fixed at 4.
(2) *Convex Hull:* Given a polygon of `N` vertices (split between Alice and Bob), convex hull is computed in an MPC. It is adapted from [**?**]. We use `N=32` for *both* experiment and `N=256` for *vec* experiment.
(3) *Count 102:* Alice has a string of length `N` of symbols, Bob has a regular expression of the form 1(0*)2, together they compute number of substrings that match the regular expression. It is adapted from [**?**]. We use `N=1024` for *both* and `N=4096` for *vec*.
(4) *Count 10:* Same as *Count 102* except now the regular expression is of the form 1(0+). Parameters are same as above.
(5) *Cryptonets Max Pooling:* Given a matrix of `rows`×`cols` elements that are split between Alice and Bob, they compute the max pooling subroutine of the cryptonet benchmark[**?**]. We use `rows=64, cols=64` for *both* experiment.
(6) *Database Join:* given two databases with `A` and `B` 2-element records, compute cross join. We use `A=B=32` for *both* and `A=B=64` for *vec*.
(7) *Database Variance* given a database of `len` records, compute variance. We use `len=512` for *both* and `len=4096` for *vec*.
(8) *Histogram:* Given `N` 5-star ratings, compute their histogram, taken from [**?**, **?**]. We use `N=512` for *both* and `N=4096` for *vec*.
(9) *Inner Product:* given two vectors, each of `N` elements, compute their inner product. We use `N=512` for *both* and `N=4096` for *vec*.
(10) *k-means Iteration:* performs the iteration subroutine of k-means database clustering operation [**?**, **?**]. Here `len1` is the size of input data, and `len2` is the number of clusters. We use `len1=32, len2=5` for *both* and `len1=256, len2=8` for *vec*.
(11) *Longest 102:* Similar to *Count 102* except that it computes the largest substring matching the regular expression. We use same parameters as *Count 102*, adapted from [**?**].
(12) *Max Distance b/w Symbols* Alice has a string of `N` symbols and Bob has some symbol `0`. The MPC computes the maximum distance between `0`s in the string. We adapted it from [**?**]. We use `N=1024` for *both* and `N=2048` for *vec*.
(13) *Minimal Points* Given a set of `N` points (split between Alice and Bob), a set of minimal points is computed i.e. there is no other point that has both a lower x and y coordinate, adapted from [**?**]. We use `N=32` for *both* and `N=64` for *vec*.
(14) *MNIST ReLU* given an input of `outer` × `inner` elements, executes the MNIST ReLU subroutine. We use `inner=512` for *both* and `inner=2048` for *vec*. `outer` is fixed at 16.
(15) *Private Set Intersection (PSI)* Alice holds set S1 with size `SA`, Bob holds set S2 with size `SB`, together they compute intersection of their sets. We use `SA=SB=128` for *both* and `SA=SB=1024` for *vec*.

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 amenability to vectorization, we divide benchmarks into 3 categories: 1) *High:* these include convex hull, cryptonets max pooling, minimal points and private set intersection. These benchmarks are highly parallelizable and see 25x to 70x speedup in BMR, and 30x to 55x in GMW protocol. 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 and biometric matching. 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, longest102 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, a 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 optimizations 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 although we do leave all the benchmarks in the table: if the transformation increases circuit depth beyond some threshold (e.g. more than 10% of the original circuit), we can reject the transformation. Note that in some settings it may still be desirable to vectorize e.g. in data constrained environments. As shown in **??**, vectorization results in reduced communication (fewer bits are transferred).

Table 6: 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 present evaluation for communication size in **??**, circuit generation time in **??**, number of gates in **??** and online time and setup time in **??** and **??** respectively.

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 **??**.

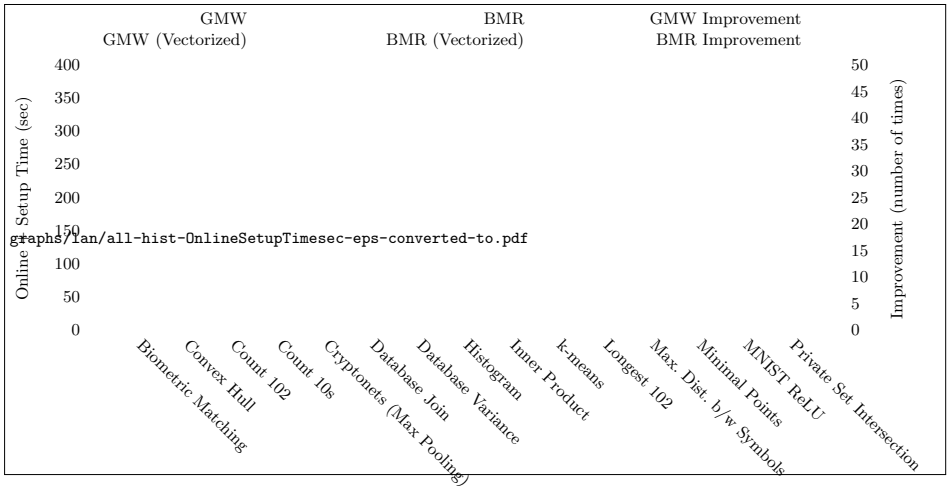


Figure 3: Circuit Evaluation Time (Setup + Online) of Benchmarks

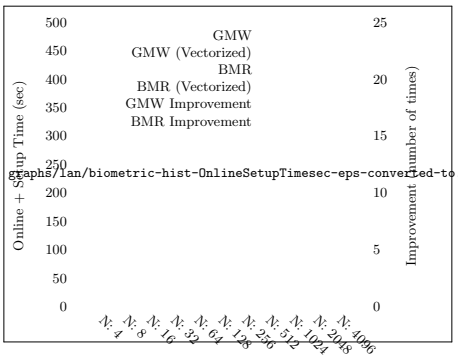


Figure 4: Biometric Matching Circuit Evaluation Time, x-axis lists database size

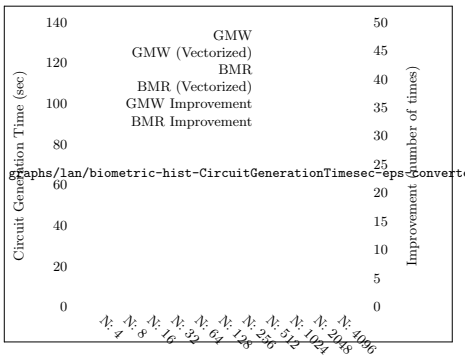


Figure 6: Biometric Matching Circuit Generation Time, x-axis lists database size

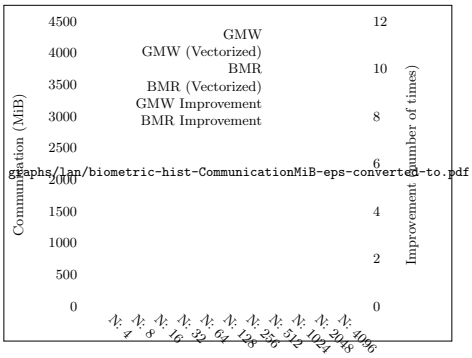


Figure 5: Biometric Matching Communication Size, x-axis lists database size

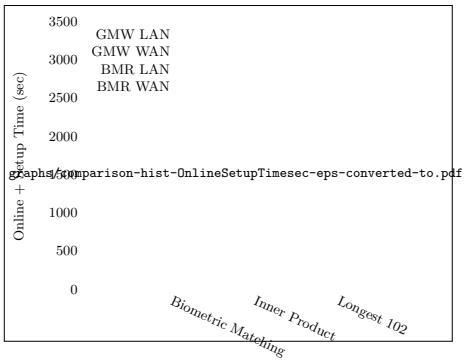


Figure 7: LAN vs. WAN: Circuit Evaluation Time Comparison

7.4 Comparison with MOTION-native Inner Product

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

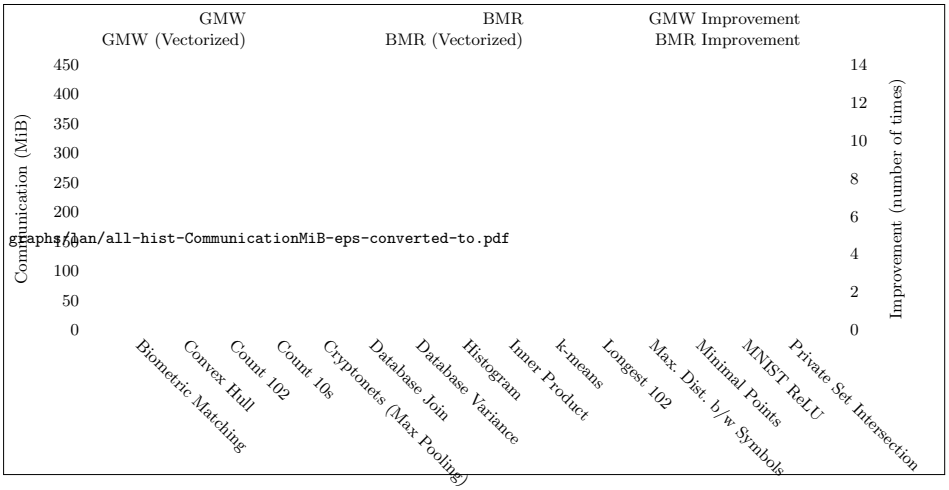


Figure 8: Communication Size of Benchmarks

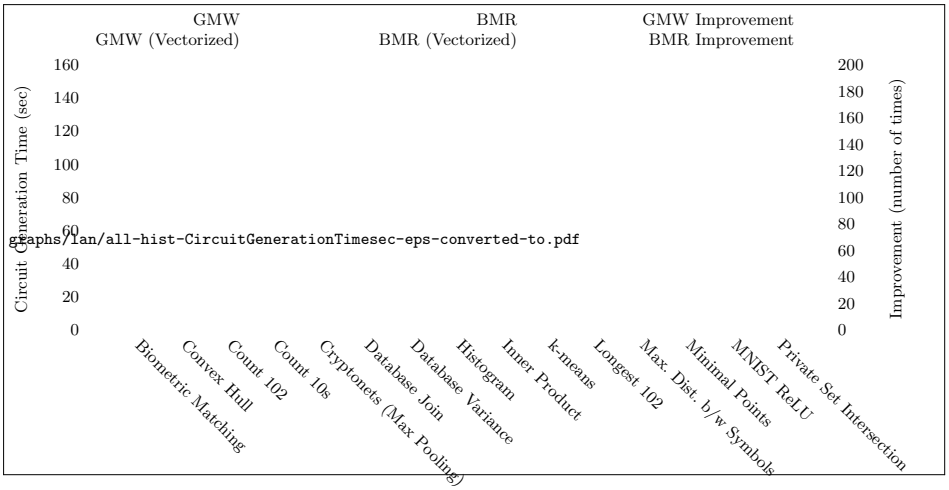


Figure 9: Circuit Generation Time of Benchmarks

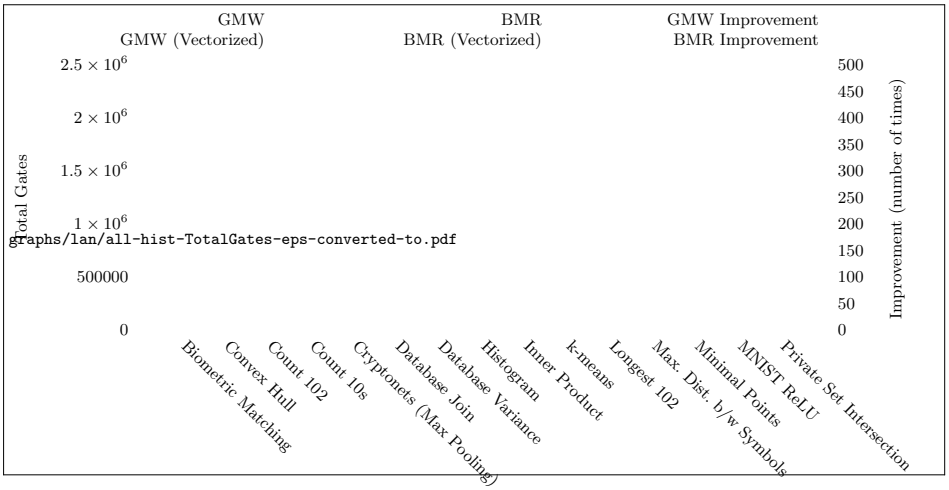


Figure 10: Number of Gates of Benchmarks

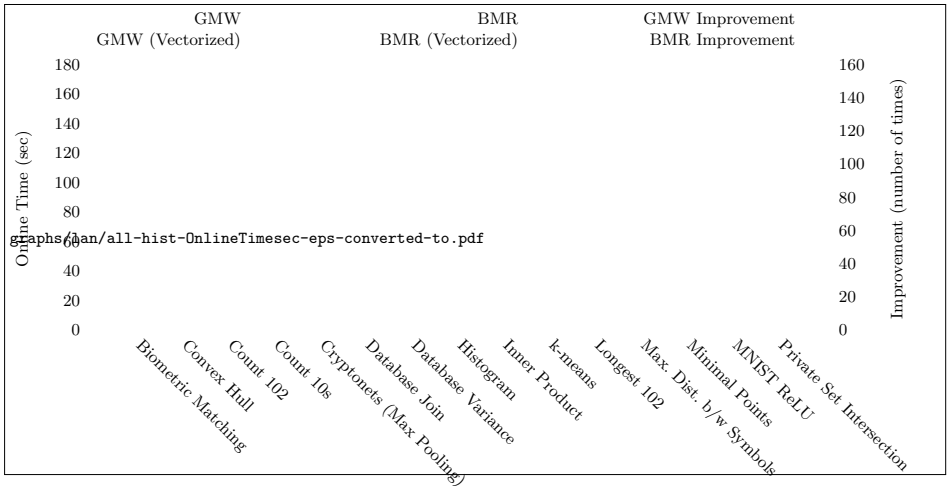


Figure 11: Online Time of Benchmarks

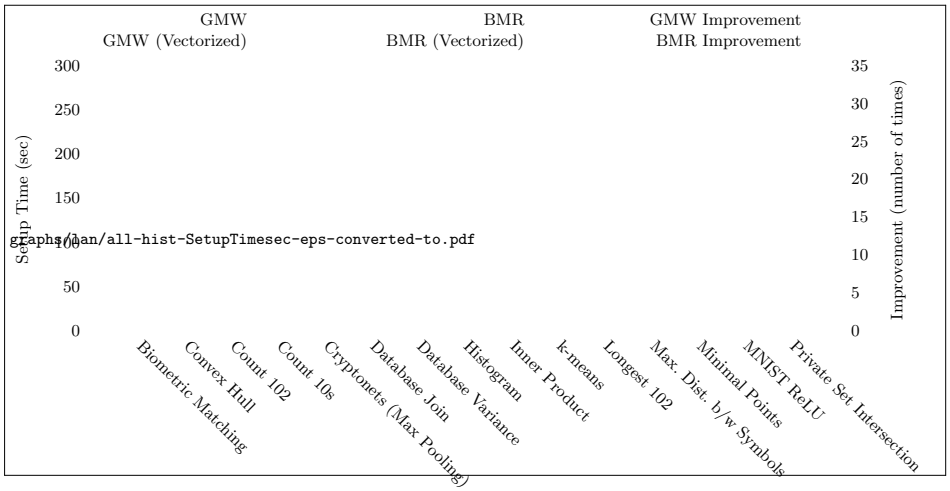


Figure 12: Setup Time of Benchmarks

and expensive in Boolean GMW). The MOTION-native loop ran result += mult_unsimdified[i]; while our loop generated and ran result[i] = result[i-1] + mult_unsimdified[i]; recall that the scalar expansion is an artifact of our vectorization. We rewrote the accumulation (manually, for testing purposes) 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. The poor performance of our loop is not a limitation of our approach, but it could be a limitation of our current implementation. Recall that the next phase 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 MOTION's static analysis will detect the associative accumulation in the scalar expansion code, it is realistic to expect that it will in the above only slightly more complex code. It still might lead to confusion in the static analysis as analysis on the AST is difficult (unfortunately, we ran out of time and did not experiment further).

We conjecture that MPC Source, a straight-forward representation, 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 as future work.

The most closely related 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 $N = 1000$; it does provide a lot of data on mixed protocol executions. Again, we estimate we are about an order of magnitude slower, which is likely due to the same issue as Inner Product — the computation of *min* can be optimized.

8 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.

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 off-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 and achieve significant speedup.

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.

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 the opportunities for future work — integration with protocol mixing, divide-and-conquer reasoning and parallelization, as well as inter-procedural

[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.

context-sensitive analysis at the level of MPC Source will improve MPC programmability and efficiency.