Compilation and Backend-Independent Vectorization for Multi-Party Computation

Benjamin Levy and Ben Sherman*[1], Muhammad Ishaq[2], Lindsey Kennard[3], Ana Milanova[4], and Vassilis Zikas[5]

[1,4]Rensselaer Polytechnic Institute
[2,5]Purdue University
[3]STR
[1]{levyb3, shermb}@rpi.edu, [2,5]{ishaqm, vzikas}@purdue.edu, [3]fireelemental.ne@gmail.com, [4]milanova@cs.rpi.edu

Abstract

Recent years have witnessed a push to bring multi-party computation (MPC) to practice and make it accessible to the end user/programmer. Despite novel ideas, on frontend language design (e.g., Wysteria, Viaduct), backend protocol design and implementation (e.g., ABY, MOTION), or both (e.g., SPDZ), classical compiler optimizations remain largely under-utilized (if not completely unused) in MPC programming. A likely reason is that such optimizations are often applied on a middle-end intermediate representation such as SSA.

We put forth a methodology for an MPC programming compilation toolchain, which by mimicking the compilation methodology of standard imperative languages enables middle-end optimizations on MPC, yielding significant improvements. To this direction we devise an MPC circuit compiler that allows MPC programming in what is essentially Python, and inherits the structure (and therefore optimization opportunities) of the classical compilation pipeline. Our key conceptual contribution is advancing an intermediate language, which we call *MPC-IR*, that can be viewed as the analogue, in an MPC program's compilation, of (enriched) SSA form. MPC-IR is a particularly appealing intermediate language as it allows backend-independent optimizations, a close analogy to machine-independent optimizations in classical compilers. Demonstrating the power of our approach, we focus on a specific backend-independent optimization, SIMD-vectorization: We devise a novel classical-compiler-inspired automatic SIMD-vectorization on MPC-IR, which we show leads to significant speedup in circuit generation time and running time, as well as significant reduction in communication size and number of gates over the corresponding iterative schedule.

We implement and benchmark our compiler from a Python-like program to an optimized circuit that can be fed into an MPC backend (for our benchmarks we make use of the MOTION backend for MPC). We view our exhaustive benchmarks as both a way to validate our optimization and end-to-end compiler, and as a contribution, by itself, to a more complete benchmarks suite for MPC programming—such benchmarks suites are common in classical compilers.

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

*joint first authros.

output (correctness) and it does not leak any information about the individual party inputs beyond what can be deduced from the output (privacy).

MPC theory dates back to the early 1980s [Yao82; GMW87; BGW88; CCD88]. Long in the realm of theoretical cryptography, MPC has seen significant advances in application in recent years. New tools and compilers bring MPC closer to practice and wider applicability, e.g., [Bog+09; BG11; MZ17; MR18]. The overarching goal of the relevant research has been to build toolchains (high-level languages, compilers, circuits, and protocol implementations) that enable non-expert programmers to write *secure* and *efficient* programs without commanding extensive knowledge of cryptographic primitives.

Recent advances in MPC programming technology tend to focus on either frontend language design (e.g., Wysteria [RHH14a] and Viaduct [Aca+21]) or backend circuit/protocol design and implementation (e.g., SPDZ family[KOS16; Ara+18; Kel20], MOTION [Bra+22]). The former, frontend-focused thread devised high-level constructs to express multiple parties, computation by different parties, and information flow from one party to another [RHH14a; Aca+21]. The latter, backend-focused thread devised cryptographic protocols, typically at the circuit-level [DSZ15; Ara+18; Bra+22; Pat+21; KOS16]. The two large categories here are gate-by-gate, i.e., GMW-style [GMW87], evaluation backends and garbled-circuits based [Yao82] constructions. (A line of work focuses on optimal combination, aka *mixing* [Büs+18; Aca+21]).

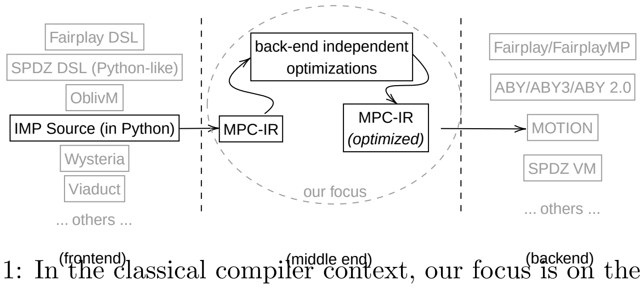

Figure 1: In the classical compiler context, our focus is on the middle-end.

In this work we focus on the *middle end*. We formalize an intermediate representation (IR) tailored to MPC, called *MPC-IR*, and focus on what we call *backend-independent* optimizations, a close analogue to *machine-independent* optimization in the classical compiler. Fig. 1 depicts our position in the compiler stack. As in classical compilers, we envision different front ends compiling into MPC-IR—the front-end used in this work is *IMP Source*, an easy to use Python-like language (cf. comparison below).

The MPC-IR exposes the *linear structure* of MPC programs, which simplifies program analysis; this is in contrast to IMP source, which has branching constructs. In the same time, MPC-IR is sufficiently "high-level" to support analysis and optimizations that take into account control and data flow in a specific program. As an added benefit, MPC-IR facilitates simple and abstract modeling of (amortized) cost associated to different operations. Indeed, this makes MPC-IR particularly suitable for optimizations such as protocol mixing [Büs+18; IMZ19; Fan+22; Esc+20; MR18] and SIMD-vectorization, which takes advantage of amortization at the circuit level. For our benchmarks and experiments we use MOTION [Bra+22] as our backend to demonstrate the advantage of optimizations at the IR level. The main reason of this choice is that MOTION already embodies different MPC paradigms (e.g., GMW and Yao style protocols), and even supports protocol-mixing which, as discussed above, can greatly benefit form our IR.[1] Nonetheless, all these optimizations are backend independent, and can be applied to alternative state-of-the-art MPC backends such as MP-SPDZ [Kel20].

Our Contribution We describe an end-to-end compiler framework that takes a Python-like routine (frontend) and produces optimized MOTION code (backend). More concretely, (a) we describe our (Python-like) IMP Source language, its syntax and semantic restrictions; (b) we provide formal specification of MPC-IR; (c) we devise a translation of IMP Source into MPC-IR, (d) we demonstrate a specific backend-independent optimization: novel SIMD-vectorization on MPC-IR; and (e) we translate MPC-IR into MOTION code.

[1]A natural next step for our work is to add automatic protocol mixing, e.g., [IMZ19], to produce highly vectorized mixed-protocol MPC programs.

We also provide an analytical model for cost estimation of amortized schedules. Our model simplifies the problem of cost estimations by abstracting away several of the relevant complexities. We note in passing that such cost modeling is important as it drives not only vectorization but also optimizations such as protocol mixing and scheduling [Büs+18]. Given the restrictions on MPC programs that are imposed by privacy requirements—e.g., the need for bounded loops—and the abstactions of our model, one might think that this problem is simpler for MPC than the classical scheduling problem (which is known to be NP-hard). One might ask:

Is cost estimation for amortized schedules in MPC (in this model) tractable?

We answer this question to the negative: We show that even in our cost model, the problem remains NP-Hard; we show this via a reduction to the Shortest Common Supersequence (SCS) problem.

The above demonstrates that optimized MPC scheduling is as an interesting problem as schedule optimization in classic compiler—a known hard problem with no exact, efficient solution. In this work, we provide such optimizations by utilizing our cost model, IR, and taking inspiration from the compilers literature attacking this problem for classical parallel and high-performance computing (HPC): a common technique there is *vectorization*, aka "SIMDification". Informally, a Single Instruction, Multiple Data (in short, SIMD) operation works with vectors of data instead of scalars, and replaces N operations on scalars with a single operation on vectors of size N. Automatic vectorization can therefore drastically improve parallel scheduling time, by identifying such "groupable" operations and replacing them with a corresponding SIMD operation in a given schedule.

In this work, we provide a novel technique for automatic vectorization of MPC-IR programs. Vectorization not only reduces running time, but also reduces communication by enabling better packing. If the underlying framework supports SIMD gates—our chosen backend does—this results in a smaller circuit and reduces circuit generation time as well. We demonstrate expressivity of the source language by running the compiler on 15 programs with interleaved if- and for-statements. For the backend MPC circuits, we produce code for Boolean GMW and BMR [BMR90] only (our chosen backend does not support all operations in Arithmetic GMW protocol and we do not yet support protocol mixing). Towards evaluation (cf. §7.3), we run experiments with two parties (2PC) and three parties (3PC). *Circuit evaluation time* in the 2PC setting shows improvement of up to 30x (40x for 3PC setting) in GMW and up to 45x (55x for 3PC) in BMR. For the operations that do not depend on number of parties, *Communication size* reduces by up to 13x in GMW and 3x in BMR. Similarly, *circuit generation time* and *number of gates* reduce, respectively, by up to 200x and 480x in GMW, and 80x and 450x in BMR.

Full source code and benchmarks, are available as open source on GitHub (link removed to preserve anonymity).

Our results emphasize the importance of backend-independent optimizations; we believe that our work can lead to future work on backend-independent compilation and optimization, ushering new MPC optimizations and combinations of optimizations.

The rest of the paper is organized as follows. §2 presents an overview of our techniques. §3 describes cost estimation model and proves NP-hardness of optimal scheduling. §4 details the front-end phases of the compiler, §5 focuses on backend-independent vectorization, and §6 has brief description of translation into MOTION. §7 presents the experimental evaluation. §8 concludes the paper with with a discussion of related work.

2 Overview of our Methodology

To demonstrate our main technical contributions, we first provide a high-level overview of our methodology, using the standard MPC benchmark of Biometric matching as our running example.

IMP-Source as MPC Source Code. An intuitive (and naive) implementation of Biometric matching is as shown in Listing 1(a). Array C is the feature vector that we wish to match and S is the database of N size-D vectors that we match against.

```
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-IR

```
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-IR

Table 1: Biometric Matching from IMP Source to Optimized MPC-IR. – MPC-IR is an SSA form without conditionals, therefore conditional on lines 13-15 in (a) turns into linear code on lines 12-16 (b). – In (c), our compiler fully vectorizes the SUB and MUL operations on lines 9 and 10 of (b). The computation of sum (line 11 in (b)) is sequential across the j-dimension, but it is parallel across the i-dimension as the loop on lines 12-16 in (c) illustrates; here p[I,j] refers to the j-th column in p.

Our compiler takes essentially standard IMP [NK14] syntax and imposes certain semantic restrictions (details will follow). 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 (secrets), however, the array sizes D and N respectively are plaintext. The code iterates over the S 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 and the corresponding sum of squares.

MPC-IR and Schedule Cost. Our compiler generates MPC-IR, a *linear* Static Single Assignment (SSA) form. Listing 1(b) shows the MPC-IR translation of the code in 1(a).

We turn to our analytical model to compute the *cost* of the iterative program. Assume cost β for a local MPC operation (e.g., XOR in Boolean sharing or ADD in Arithmetic sharing) and cost α for a remote MPC operation (e.g., MUX, CMP, etc.). Assuming that ADD is β and SUB, CMP and MUX are α, the MPC-IR in Listing 1(b) gives rise to an iterative schedule with cost $ND(2\alpha + \beta) + N(3\alpha)$.

Vectorized MPC-IR and Schedule Cost. We can compute all $N * D$ subtraction operations at line 9 in 1(b) in a single SIMD instruction; similarly we can compute all multiplication operations at line 10 in a single SIMD instruction. Our compiler runs Listing 1(b) through the vectorization optimization to produce 1(c). Note that this is still our IR, Optimized MPC-IR. The compiler turns this code into variables, loops and SIMD primitives (if supported), suitable for the backend to generate the circuit.

In MPC backends, executing n operations "at once" in a single SIMD operation costs less than executing those n operations one by one. This is particularly important for interactive gates, since it allows many 1-bit values to be sent at once. We consider that each operation has a *fixed* portion that benefits from amortization and a *variable* portion that 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, $2\alpha_{\mathit{fix}} + D\beta_{\mathit{fix}} + N(3\alpha_{\mathit{fix}}) << ND(2\alpha_{\mathit{fix}} + \beta_{\mathit{fix}}) + N3\alpha_{\mathit{fix}}$. Empirically, we observe orders of magnitude improvement e.g., for Biometric Matching evaluation time, 10x and 23x in GMW and BMR respectively in 2PC, and 12x and 28x in 3PC. 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 (Parallel) Cost Model

Next, we introduce our model for cost estimation of the MPC schedules and prove that optimal schedule (of MPC) is NP-Hard.

3.1 Scheduling in MPC

We work in a single protocol setting i.e., all MPC tasks are evaluated in a single protocol from start to finish. In addition, we abstract common features of MPC execution, in the following assumptions:

(1) There are two types of MPC instructions, local and remote. A local instruction (i.e., ADD or XOR) has cost β and a remote instruction (i.e. MUL) has cost α, where $\alpha >> \beta$. We assume that all remote instructions have the same cost α and all local ones have the same cost β.

(2) 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 $f(n) = \alpha_{\mathit{fix}} + n\alpha_{var}$ in contrast to unamortized execution $g(n) = n\alpha_{\mathit{fix}} + n\alpha_{var}$. We have $\alpha_{\mathit{fix}} << n\alpha_{\mathit{fix}}$ and since fixed cost dominates variable cost (particularly for remote operations), we have $f(n) << g(n)$.

(3) 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 $\alpha_{\mathit{fix}} + 2\alpha_{var}$, however a MUL and a MUX instruction still cost $2\alpha_{\mathit{fix}} + 2\alpha_{var}$ even when scheduled "in parallel".[2]

3.2 (Intractability of) Optimal MPC Scheduling

Given a serial schedule (a linear graph) of an MPC program i.e. a sequence of instructions $S := (S_1; \ldots; S_n)$, where $S_i \in \{B, A\}, 1 \leq i \leq n$, 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 instructions of the same kind.

(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 of P follows due to the preservation of def-use *dependencies*. One can easily argue by induction on the length of schedule S that the computed function is the same in both S and P.

The cost of schedule S is

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

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 §2.) The cost of schedule P is more interesting:

[2]This is not strictly true, but assuming it, e.g. as in [IMZ19; DSZ15; MR18], helps simplify the problem.

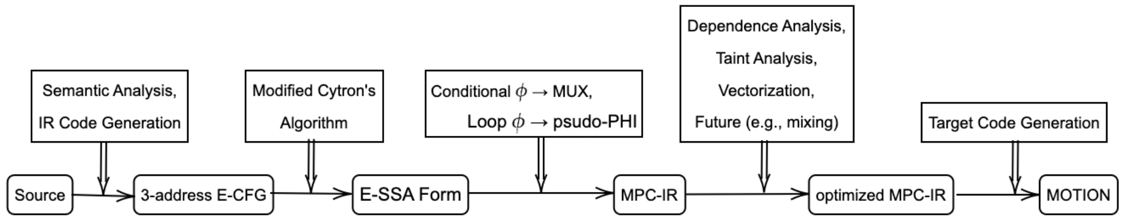

Figure 2: Compiler Framework.

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

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 §2.)

Our goal is to construct a parallel schedule P that reduces the program cost (when compared to cost of S). One would hope that simpler MPC program structure would make optimal schedule tractable. Intuitively, the problem is to combine multiple independent schedules (or sequences of instructions) into a single schedule where same instructions are scheduled into a SIMD-instruction P_i. This amounts to finding a Shortest Common Supersequence for the independent schedules. We formalize the argument in §9.1 and show that the scheduling problem is NP-hard via a reduction to the Shortest Common Supersequence problem [Vaz10].

4 Compiler Frontend

This section presents an overview of our compiler, followed by our source syntax and semantic restrictions.

4.1 Overview

Fig. 2 shows the phases of our compiler. A key novelty of our work is the systematic translation of high-level source, which includes if-then-else statements, into a linear sequence of primitive MPC instructions. We start with an Abstract Syntax Tree (AST) source syntax, then convert it into a Control-flow graph (CFG), as Cytron's classical SSA algorithm [Cyt+91] is defined over a CFG. SSA is the natural means for converting if-then-else statements into MUX primitives, as there is correspondence between ϕ-nodes and MUX primitives (cf. §10.1). Yet, to our knowledge, we are the first to present an algorithm that uses Cytron's SSA conversion and therefore takes advantage of the properties of Cytron's SSA, particularly a *minimal number* of phi-nodes that results in a minimal number of MUX primitives. We then translate into MPC-IR, which is most conveniently described with an AST syntax (cf. §5.4 and §10.2).

We also note that we settled on this process after the straightforward approach failed. Initially, we attempted to reuse existing implementations of SSA intermediate representations such as Soot Shimple (`http://soot-oss.github.io/soot/`) or LLVM IR (`https://llvm.org/`). The problem was that these were CFG representations and they had lost connection between the ϕ-node and the conditional that triggered the ϕ-node. Specifically, given $\mathtt{x} = \phi(\mathtt{y}, \mathtt{z})$, there is no information what conditional triggered the ϕ-node and whether y corresponds to the true or false branch of the conditional. Moreover, ϕ-nodes with 3 or more arguments are common in standard SSA. Furthermore, existing SSA IRs do not handle arrays, while handling of arrays is important for vectorization. As a remark, while it is possible to reconstruct the missing information via a form of control dependence analysis, and it is possible to add arrays, this proves very difficult due to the complexity of the IRs (these IRs are designed to handle richer and more complex syntax than MPC). A clean state solution, where we start from the AST and retain all necessary information in the CFG i.e. enhanced-CFG (or E-CFG in Fig. 2) and construct enhanced-SSA (or E-SSA) that handles arrays, proves the correct choice and drives our progress on the compiler. The E-SSA form naturally gives rise to

MPC-IR, where conditional ϕ-nodes translate to MUX nodes, and loop ϕ-nodes translate into what we call pseudo PHI-nodes (shown in lines 4 and 12 in Listing 1(b)).

4.2 Syntax and Semantic Restrictions

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

$$
\begin{array}{ll}
e ::= e \; op \; e \mid \mathsf{x} \mid \texttt{const} \mid \texttt{A[}e\texttt{]} & \textit{expression} \\
s ::= s; s \mid & \textit{sequence} \\
\quad \mathsf{x} = e \mid \texttt{A[}e\texttt{]} = e \mid & \textit{assignment stmt} \\
\quad \texttt{for } i \texttt{ in range(}I\texttt{)} : s \mid & \textit{for stmt} \\
\quad \texttt{if } e\texttt{: } s \texttt{ else: } s & \textit{if stmt}
\end{array}
$$

The syntax allows for array accesses, arbitrarily nested loops, and if-then-else control flow.

The prototype version of the compiler assumes the following semantic restrictions on source programs. Currently, the compiler does not enforce the restrictions, however, they can be easily encoded as rules in a syntax-directed translation [ASU86; Sco09] over the syntax above. The reason why we do not implement the rules is because the majority of these restrictions are implementation restrictions that can be lifted in future versions of our compiler.

- 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.
- Arrays are one-dimensional. N-dimensional arrays are linearized and accessed in row-major order.
- Array subscripts are plaintext. We will lift this restriction in future work by applying the standard linear scan [Liu+15; Ara+18] when the subscript is a secret-shared value.
- 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., $\mathsf{A}[i, j, k] = ...$ where i, j and k are the indices of the *outermost* loops enclosing the array write statement. These indices loop over the three dimensions of A and all write accesses to A follow this restriction. We note that this is a restriction on the vectorization optimization; there is no reason to restrict arbitrary writes in the code, they just are not optimized. We state this restriction upfront as it simplifies vectorization and its exposition.
- The final restriction disallows array writes from within if-then-else statements. This is to ensure that arguments of MUX in the MPC-IR translation are base types, i.e., just int or bool. In our experience, this causes a minor inconvenience to the programmer as they may not write

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

  Instead they write

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

In addition, our compiler defines and implements a taint type system at the level of MPC-IR. We define the base MPC-IR syntax and the type system in §10.3. We note that while the programmer writes annotations at the level of IMP Source (as in Listing 1(a)), the annotations propagate through the transformations; annotations are inferred and checked with a standard taint analysis (based on the type system) at the level of MPC-IR. The only required annotations are on the input arguments.

For the rest of this paper we write i, j, k to denote the loop nesting: 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 $\mathsf{A}[i, j, k]$ to denote *canonical access* to an element, either array element or a scalar expanded to its loop nest i, j, k. To simplify the presentation we describe our algorithms in terms of three-element tuples i, j, k, however, discussion generalizes to arbitrarily large loop nests.

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 [AK87], our algorithm is unique as it works on the MPC-IR SSA-form representation. We posit that vectorization over MPC-IR is a problem that warrants a fresh look, in part because of MPC's unique linear structure and in part because vectorization interacts 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., [DSZ15; Büs+18; IMZ19]).

5.1 Dependence Analysis

We build a dependence graph where the nodes are the MPC-IR statements and the edges represent the def-use relations. Since MPC-IR is an SSA form, def-use edges $X \to Y$ are explicit. We distinguish between *forward* edges where X appears before Y in the linear MPC-IR and *backward* edges where Y appears before X.

Def-use edges We classify def-use edges as follows:

- 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-IR in Listing 1 is a same-level edge. A same-level edge can be a backward edge in which case a PHI-node is the target of the edge. E.g., 15 to 4 in Biometric is a same-level backward 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 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, 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 backward 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 $\mathsf{x}' = \mathsf{x}$ 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 and Array 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.

Raise dimension The *raise_dim* function expands a scalar (or array). There are two versions of *raise_dim*. One reshapes an arbitrary access into a canonical read access in the corresponding loop. It takes the original array, the original access pattern function $f(i,j,k)$ in loop nest i,j,k and the loop bounds $((i:I),(j:J),(k:K))$ (cf. 4.2):

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

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, $\mathsf{C}' = \mathit{raise\_dim}(\mathsf{C}, j, (i:N, j:D))$ lifts the 1-dimensional array C into a 2-dimensional array. The i,j loop now accesses C' in the canonical way, $\mathsf{C}'[i,j]$.

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; this version reduces to the above version by adding the access pattern function, which is just i:

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

Drop dimension *drop_dim* is carried out when an expanded scalar (or 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:

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

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:

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

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

- Scalars: We expand scalars 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-IR 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.
- Read-write arrays: Writes are canonical (by restriction) but reads can be non-canonical. 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. The restriction to canonical writes essentially reduces the case of arrays to the case of scalars, simplifying vectorization and correctnss reasoning.

5.3 Basic Vectorization 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 loop 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 could allow vectorization in intermediate statements in the loop. Second, it creates opportunities for vectorization in the presence of write arrays, even though Cytron's SSA adds a backward edge to the array PHI-node, thus killing vectorization of statements that access the array.

The code in blue color in the algorithm 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 §5.5. (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; thus, we elide it from this presentation.

```
{ Phase 1: Raise/drop dimension of scalars to corresponding loop nest. We traverse stmts linearly in MPC-IR. }
for each MPC $stmt$: $\mathsf{x} = Op(\mathsf{y}_1, \mathsf{y}_2)$ in loop $i, j, k$ do
  for each argument $\mathsf{y}_n$ do
    case $stmt'$(def of $\mathsf{y}_n$) $\rightarrow$ $stmt$(def of $\mathsf{x}$) of
      same-level: $\mathsf{y}'_n$ is $\mathsf{y}_n$
      outer-to-inner: add $\mathsf{y}'_n[i,j,k] = raise\_dim(\mathsf{y}_n)$ at $stmt'$ (more precisely, right after $stmt'$)
      inner-to-outer: add $\mathsf{y}'_n[i,j,k] = drop\_dim(\mathsf{y}_n)$ at $stmt$ (more precisely, in loop of $stmt$ right after loop of $stmt'$)
  end for
  { Optimistically vectorize all. $I$ means vectorized dimension. }
  change to $\mathsf{x}[I,J,K] = Op(\mathsf{y}'_1[I,J,K], \mathsf{y}'_2[I,J,K])$
end for
{ Phase 2: Recreating for-loops for cycles; vectorizable stmts hoisted up. }
for each dimension $d$ from highest to 0 do
  for each PHI-node $n$ in loop $i_1, ..., i_d$ do
    compute $closure(n)$
  end for
  { $cl_1$ and $cl_2$ intersect if they have common statement or update same array; "intersect" definition can be expanded }
  while there are closure $cl_1$ and $cl_2$ that intersect do
    merge $cl_1$ and $cl_2$
  end while
  for each closure $cl$ (after merge) do
    create for $i_d$ in ... loop
    add PHI-nodes in $cl$ to header block
    add target-less PHI-node for $\mathsf{A}$ if $cl$ updates array $\mathsf{A}$
    add statements in $cl$ to loop in 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 $\mathsf{A}_1$ = PHI($\mathsf{A}_0$,$\mathsf{A}_k$) do
    in vectorizable stmts, replace use of $\mathsf{A}_1$ with $\mathsf{A}_0$
    discard PHI-node if not used in any $cl$, replacing $\mathsf{A}_1$ with $\mathsf{A}_0$ or $\mathsf{A}_k$ as necessary
  end for
end for
{ Phase 3: Remove unnecessary dimensionality.}
```

Consider our running example in Listing 1(B). Phase 1 will raise dimensions of `min_sum!1` to a 1-dimensional array as it is defined outside of the loop but is used inside the i-loop. It will expand `C` into a 2-dimensional (i,j)-array. Phase 1 will also add *drop_dim* to drop the dimension of `sum!3`, which is defined in the inner loop and is of dimension (i,j), but is used in the outer i-loop and needs to align to that loop

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

Figure 3: MPC-IR Syntax and Semantics. γ defines the semantics of MPC-IR which is a linearization of input MPC-IR. A SIMD operation parallelizes operations across the vectorized J dimension. $\mid\mid$ denotes parallel execution, which is standard. γ of a for loop unrolls the loop. ; denotes sequential execution. Iterative MPC-IR trivially extends to non-vectorized dimensions over the enclosing loops.

dimensionality.

Phase 2 starts with the inner j-loop. There are no dependences for the SUB and MUL statements (lines 9-10 in Listing 1(B)) and they are moved outside of the loop. The ADD is part of a cycle and it remains enclosed in a j-loop. Moving up to the outer i-loop, the addition j-loop is not part of a cycle in i and Phase 2 moves that loop outside vectorizing the i dimension of the summation (this results in the loop in lines 12-16 in Listing 1(C)). The MUX computations are part of cycles and they remain in i-loops.

5.4 Correctness Argument

We build a correctness argument as follows. First, we define the MPC-IR syntax. We then define the *linearization* of an MPC-IR program as an *interpretation* over the syntax. The linearization is a *schedule* as defined in §3. We prove a theorem that states that the Basic vectorization algorithm preserves the def-use relations, or in other words, linearization of the vectorized MPC-IR program gives rise to the exact same set of def-use pairs as linearization of the original program does. It follows easily that the schedule corresponding to the vectorized program computes the same result as the schedule corresponding to the original program.

MPC-IR Syntax Fig. 3 states the syntax and linearization semantics of MPC-IR. Although notation is heavy, the linearization simply produces schedules as discussed in §2 and §3. The iterative MPC-IR gives rise to what we called sequential schedule where loops are unrolled and MPC-IR 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 (with our restriction on array writes to canonical writes). $\mathsf{x}[i,J,k]$ 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. There are semantic restrictions over the syntax: (1) x is a 3-dimensional array and (2) $\mathsf{x}[i,J,k]$ is enclosed in for-loops on non-vectorized dimensions i and k:

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

Linearization Linearization is the concretization operation, which, as we mentioned earlier computes a schedule. The concretization function γ is defined as an interpretation of MPC-IR syntax, as is standard. It is shown in the middle column of Fig. 3. The concretization of an $\mathsf{op\_SIMD}$ statement expands the vectorized dimension(s) into *parallel* statements; $\mid\mid$ introduces SIMD (parallel) execution. The concretization of the $\mathsf{for}\ i\ \mathsf{in\ range}(I) : s$ statement simply unrolls the loop substituting i with 0, 1, etc.; here ; denotes *sequential* execution.

As an example, consider the vectorized MPC-IR from our running example. All variables are two dimensional arrays and the loop is vectorized in I but iterative in 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])
```

Assuming $D = 2$ and $I = 2$ for simplicity, linearization produces the following schedule:

```
sum!3[0,0] = PHI(sum!2[0,0], sum!4[0,−1]) ||
                         sum!3[1,0] = PHI(sum!2[1,0], sum!4[1,−1])
;
sum!4[0,0] = ADD(sum!3[0,0], p[0,0]) ||
                         sum!4[1,0] = ADD(sum!3[1,0], p[1,0])
;
sum!3[0,1] = PHI(sum!2[0,1], sum!4[0,0]) ||
                         sum!3[1,1] = PHI(sum!2[1,1], sum!4[1,0])
;
sum!4[0,1] = ADD(sum!3[0,1], p[0,1]) ||
                         sum!4[1,1] = ADD(sum!3[1,1], p[1,1])
```

Note that by definition of the pseudo PHI function, `PHI(sum!2[0,0], sum!4[0,-1])` evaluates to `sum!2[0,0]` and therefore, the -1 index in the second argument does not matter.

Statements and def-uses over MPC-IR Let a be an MPC-IR program. Since MPC-IR is an SSA form, def-use edges in a are explicit (as in §5.1): if $s_0 \in a$ defines variable x, e.g., x = ..., $s_1 \in a$ uses x, e.g., ... = ...x, then there is a def-use edge from s_0 to s_2. We write $s_0[i, j, k]$ for statement s_0 enclosed in loop nest i, j, k.

Let a_0, a_1 be two MPC-IR programs. Two statements, $s_0 \in a_0$ and $s_1 \in a_1$ are *same*, written $s_0 \equiv s_1$ if they are of the same operation and they operate on the same variables: same variable name and same dimensionality. Recall that dimensions in MPC-IR are either iterative, lower case, or vectorized, upper case. Two statements are same even if one operates on an iterative dimension and the other one operates on a vectorized one, e.g., $s_0[i, j, k] \equiv s_1[I, j, K]$.

Statements and def-uses over linearized schedule An *atomic* statement is a statement produced by linearization. We write $\underline{s_0}$ to denote statements in the concrete schedule as well as $\underline{s_0}[\underline{i}, \underline{j}, \underline{k}]$ to denote fully instantiated values of i, j, and k, such as for example $\underline{s_0}[0, 1, 0]$. Clearly, the linearization of same statements produces the same set of atomic statements in the linearized schedule.

A def-use pair of atomic statements, denoted $\underline{s_0} \to \underline{s_1}$ (indexing implicit), is defined in the standard way as well: $\underline{s_0}$ writes a location, say x$[\underline{i}, \underline{j}, \underline{k}]$, and $\underline{s_1}$ reads the same location.

Formal treatment Property P defined below relates the linearized schedule of iterative MPC-IR program a_0 to the linearized schedule of the vectorized program a_1. More precisely, a_0 is the MPC-IR program augmented with raise and drop dimension statements, i.e., Phase 1 without optimistic vectorization of all dimensions. a_1 is produced from a_0 by Phase 2 of the Basic Vectorization algorithm.

Definition 1. *We say that $\gamma(a_0) \equiv \gamma(a_1)$ iff (1) atomic statement $\underline{s}[\underline{i}, \underline{j}, \underline{k}] \in \gamma(a_0)$ iff $\underline{s}[\underline{i}, \underline{j}, \underline{k}] \in \gamma(a_1)$ and (2) $\underline{s_0} \to \underline{s_1} \in \gamma(a_0)$ iff $\underline{s_0} \to \underline{s_1} \in \gamma(a_1)$ (indexing implicit).*

The main theorem below states that Basic Vectorization preserves def-use edges. We extend a more detailed argument, albeit standard, in Section 9.2:

Theorem 1. *$\gamma(a_0) \equiv \gamma(a_1)$.*

The key argument is that the Basic Vectorization algorithm preserves def-uses when it transforms a_0 into a_1. This leads to preservation of concrete edges in $\gamma(a_0)$ into $\gamma(a_1)$. A corollary of the main theorem follows (we prove it in Section 9.3):

Corollary 1.1. $\gamma(a_0)$ and $\gamma(a_1)$ produce same result, or more precisely, for every location x$[\underline{i}, \underline{j}, \underline{k}]$, $\gamma(a_0)$ and $\gamma(a_1)$ compute the same result.

5.5 Extension with Array Writes

Array writes may introduce infeasible loop-carried dependencies. Consider an example from [AN88]:

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

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

```
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]);
```

`B_1 =PHI(B_0,B_2)` anchors a cycle that includes statement `A_2 =update(A_1, i, B_1[i] + 10)`; a naive approach will not vectorize the latter statement even though 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 $0 \leq j < J$ nested at i, j, k. Here i is the outermost loop and k is the innermost loop.

```
for each array A written in loop $j$ do
  { including enclosed loops in $j$ }
  dep = False
  for each def-of-A: $\mathsf{A}_m[f(i,j,k)]=...$ and use-of-A: $...=\mathsf{A}_n[f'(i,j,k)]$ in loop $j$ do
    if $\exists \underline{i}, \underline{j}, \underline{j}', \underline{k}, \underline{k}'$, s.t. $0 \leq \underline{i} < I$, $0 \leq \underline{j}, \underline{j}' < J$, $0 \leq \underline{k}, \underline{k}' < K$, $\underline{j} < \underline{j}'$, and $f(\underline{i}, \underline{j}, \underline{k}) = f'(\underline{i}, \underline{j}', \underline{k}')$ then
      dep = True
    end if
  end for
  if dep == False then
    remove back edge into A's $\phi$-node in loop $j$.
  end if
end for
```

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.

We use Z3 [MB08] to check satisfiability of the formula

$$(0 \leq \underline{i} < I) \wedge (0 \leq \underline{j}, \underline{j}' < J) \wedge (0 \leq \underline{k}, \underline{k}' < K) \wedge (\underline{j} < \underline{j}') \wedge \\ f(\underline{i}, \underline{j}, \underline{k}) = f'(\underline{i}, \underline{j}', \underline{k}')$$

Formulas f and f' are simple as loop nests are typically of depth 2-3. Therefore, Z3 completes the process instantly.

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 backward edge from 6 to the phi-node 2. Analogously, we remove the backward 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 backward 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.

This step renders some array phi-nodes *target-less*, or in other words, these nodes are not targets of any def-use edge. We handle target-less phi-nodes with a minor extension of Basic Vectorization (Phase 2, extension shown in blue). First, we merge closures that update the same array. This simplifies handling of array PHI-nodes: if each closure is turned into a separate loop, each loop will need to have its own array phi-node to account for the update and this would complicate the analysis. Second, we add the target-less node of array A back to the closure that updates A — the intuition is, even if there is no loop-carried dependence from writes to reads on A, A is written and the write (i.e., update) cannot be vectorized due to a different cycle; 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 (the last snippet in blue).

Recall that we restrict array updates to *canonical updates*, that is, an update $\mathsf{A}[i,j] = ...$ is enclosed in loops on i and j. It may be enclosed into a nested loop i, j, k, however, the indices correspond to the outermost loops. This restriction ensures that the array shape does not change and raise dimension and drop dimension can be applied in the same away as in the basic case, thus allowing us to extend correctness reasoning from the basic case. We will look to relax the restriction in future work.

6 Compiler Backend

To provide an end-to-end compiler, we take the optimized MPC-IR and generate C++ code for MOTION framework. This requires overcoming a few challenges, e.g., MPC-IR requires `shared` qualifiers only on input variables, while MOTION requires all variables to be either `shared` or `plain`. To resolve this, we infer qualifiers for all variables by performing *taint analysis* (see §12.1) according to the rules of §4.2. Another challenge is that of the public values e.g., constants. Since our chosen back-end lacks supports for these, one has to provide such public values as shared input from one of the parties. Input gates are expensive and a naive implementation could introduce significant performance hit. Instead, we take a smarter approach by keeping shared copies of plain variables, and updating them in lock-step with updates to plain variable. Then, whenever the plain variable is needed in a shared context, we use the shared copy. Please see detailed treatment of the backend translation in §12.

7 Evaluation and Analysis

We performed extensive evaluation of our framework. We present results for 15 benchmarks. This is, to our knowledge, the largest set of benchmarks in this area. Moreover, we evaluated these benchmarks in 2 party computation (2PC) and 3 party computation (3PC). We show that our optimization does not depend on a specific number of parties. Further, we evaluated in both LAN and WAN settings to show further evidence of versatility of our framework; it is network agnostic. We begin by describing experiment setup in §7.1, then we enumerate our benchmark-set in §7.2 and finally, analyze the results in §7.3.

7.1 Experiment Setup

We evaluate our framework in 2 party computation (2PC) and 3 party computation (3PC) setting. Experiment hardware was generously provided by CloudLab [Dup+19]. We consider two network settings, namely the Local Area Network (LAN) and the Wide Area Network (WAN). In the LAN setting, we use `c6525-25g` machines connected via a 10Gbps link with $<$ 1ms latency. These machines are equipped with 16-core AMD 7302P 3.0GHz processors and 128GB of RAM. This setting reflects typical LAN use-case considering that 10Gbps LAN is increasingly common in business networks and even in some home networks. In the WAN setting we only performed 2PC experiments to save time, as evidenced by LAN experiments in §7.3, 3PC experiments would only take longer to run. We 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. The standard deviation, shown as error bar on top of the histogram bars in the graphs, in all observations is at most 4.5% of the mean. Therefore, more runs will not significantly improve results' accuracy.

7.2 Benchmarks

In the following discussion, the label *both* means that the specific experiment configuration is run in both non-vectorized and vectorized versions, *vec* means only vectorized version was run. For each problem, we run benchmark experiment with increasing sizes of input e.g. we ran biometric matching with database size N of$\{2, 4, 8, \dots, 4096\}$; At some input size 2^k (e.g. $N = 2^8$ in biometric matching), the non-vectorized version runs out of memory. From this point on, we run only *vec* experiment up to input size of $2^{2(k-1)}$ e.g. in case of biometric matching, we run the *vec* experiment up to database size $N = 2^{12}$. The *vec* experiment completes without issues (e.g. running out of memory) for input size of $2^{2(k-1)}$ for all benchmarks that are amenable to vectorization (see e.g., Fig. 5). While the numbers for all runs are not shown for space reasons, the run times are largely consistent, e.g. if non-vectorized experiment fails at some input size 2^k, and *vec* experiment takes X seconds to complete for the input size 2^{k-1}, then for an input size $2^{k+\ell-1}$, it takes roughly $\ell \cdot X$ seconds to complete.

We use 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 [FN21]. We use N=32 for *both* experiment and N=256 for *vec* experiment.

(3) *Count 102:* Alice has a string of N 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 [FN21]. 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+).

(5) *Cryptonets Max Pooling:* Given an $\mathtt{R} \times \mathtt{C}$-matrix with elements split between Alice and Bob, they compute the max pooling subroutine of the cryptonet benchmark[Dow+16]. We use R=64, C=64 for *both* experiments.

(6) *Database Join:* given two databases with `A` and `B` containing 2-element records, compute cross join. We use `A=B=32` for *both* and `A=B=64` for *vec.*

(7) *Database Variance:* compute variance in a database of N records. N=512 for *both* and N=4096 for *vec.*

(8) *Histogram:* Given N 5-star ratings, compute their histogram, taken from [IMZ19; FN21]. We use N=512 for *both* and N=4096 for *vec.*

(9) *Inner Product:* Compute inner product of two N-element vectors. N=512 for *both* and N=4096 for *vec.*

(10) *k-means Iteration:* iteration of k-means database clustering [JW05; VC03]. Here L is the size of input data, and N is the number of clusters. We use L=32, N=5 for *both* and L=256, N=8 for *vec.*

(11) *Longest 102:* As *Count 102* except that it computes the largest substring matching the regular expression, from [FN21]. We use same parameters as *Count 102*.

(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 [FN21]. 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 [FN21]. We use N=32 for *both* and N=64 for *vec*.

(14) *MNIST ReLU* given an input of $O \times I$ elements, executes the MNIST ReLU subroutine. We use I=512 for *both* and I=2048 for *vec*. O is fixed at 16.

(15) *Private Set Intersection (PSI)* Alice holds a set of size S_A, Bob holds a set of size S_B, they compute intersection of their sets. We use S_A=S_B=128 for *both* and S_A=S_B=1024 for *vec*.

Next, we analyze the results. Note that we fit maximum possible evidence here, going so far as to present graphs at the lowest readable size. The full result-set still does not fit unfortunately. Specifically, we do not show any 3PC graphs here, and for 2PC, we only show a subset of graphs to show trends. A few more, but still not all, graphs appear in appendix 13. In a nutshell, the experiments for 3 parties provide confirmation that adding more parties to a secure computation increases resource requirements; benefits from vectorization are even more pronounced. An interested reader may find the complete experimental results in the full version of this paper.

7.3 Results Analysis

A detailed summary of the effects of vectorization on various benchmarks is presented in Table 2. We show circuit evaluation times in Fig. 4. 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 47x to 21x speedup in BMR, and 33x to 23x 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 24x to 5x in BMR, and 24x to 3x 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). Very few operations in these programs are parallelizable, thus the speedup is lower. We see a speedup from 2x to 1.1x in BMR. In GMW, database join, count 102 and count 10s see speedup from 1.3x to 1.1x. However, longest102 and max distance between symbols suffer a slowdown of 0.5x. There is some opportunity for vectorization in these benchmarks according to our analytical model, particularly, there is a large EQ operation that is vectorized, although a large portion of the loop cannot be vectorized. We observe that transformation to vectorized code increases multiplicative depth and, the negative effect of increased depth is more noticeable in a round-based protocol like GMW. 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 reject the transformation. Note that in some settings it may still be desirable to vectorize e.g. in data constrained environments.

As shown in Fig. 9, vectorization reduces communication, up to 12x less in GMW, 3x less in BMR. We discuss the detailed reasoning for this in §13.1. The summary is that vectorization enables better packing, it effects interactive protocols like GMW more than a constant round protocol like BMR. Fig. 11 shows that vectorization reduces gates-count up to 480x in GMW, 450x in BMR. Consequently, in the highly vectorizable benchmarks, circuit generation time (see Fig. 10) for vectorized circuits is a fraction of non-vectorized circuit

Table 2: Vectorized vs Non-Vectorized Comparison in 2PC LAN setting, times in seconds, Communication in MiB, Numbers in 1000s and rounded to nearest integer; vectorized benchmarks have postfix (V) in their name.

	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

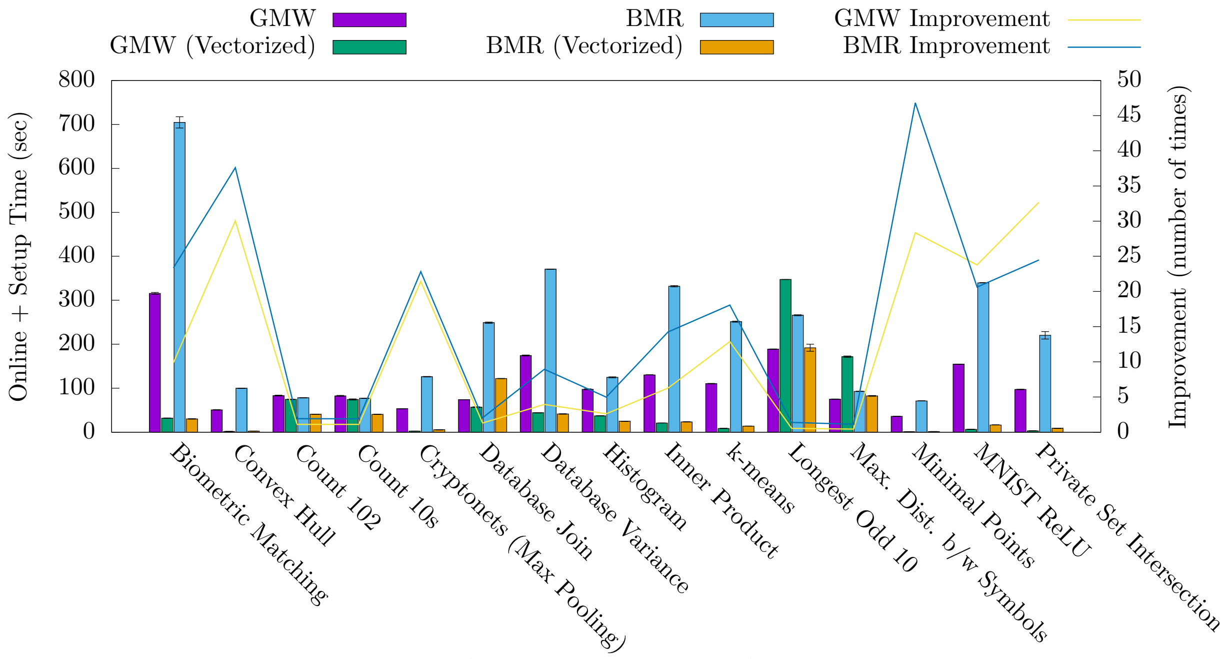

Figure 4: 2PC: Circuit Evaluation Time (Setup and Online Phase) in Seconds, LAN Setting. The Error-bars are Standard Deviation.

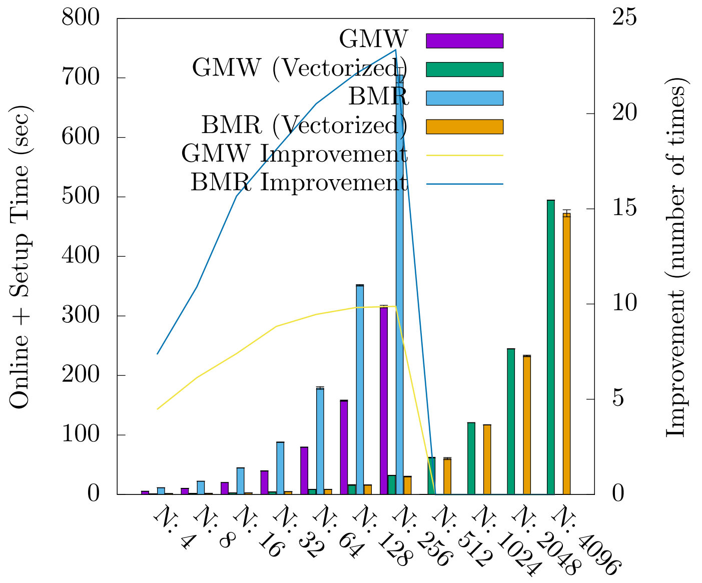

Figure 5: 2PC: Biometric Matching Eval Time, x-axis is DB-size

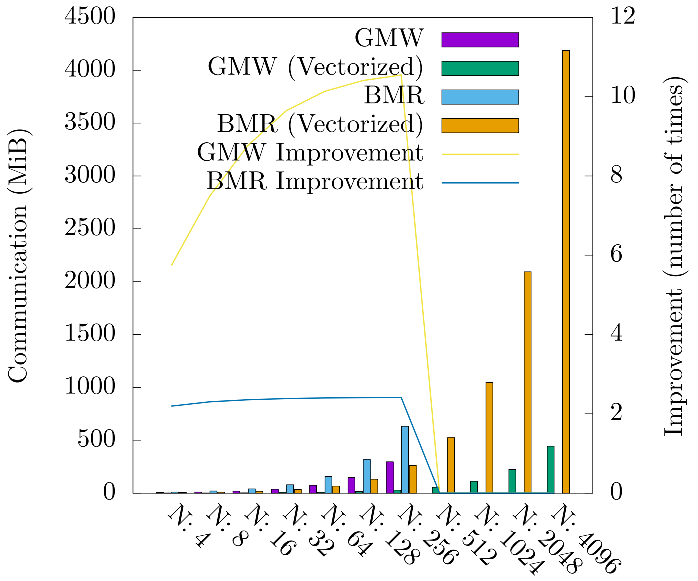

Figure 6: 2PC: Biometric Matching Comm Size, x-axis is DB-size

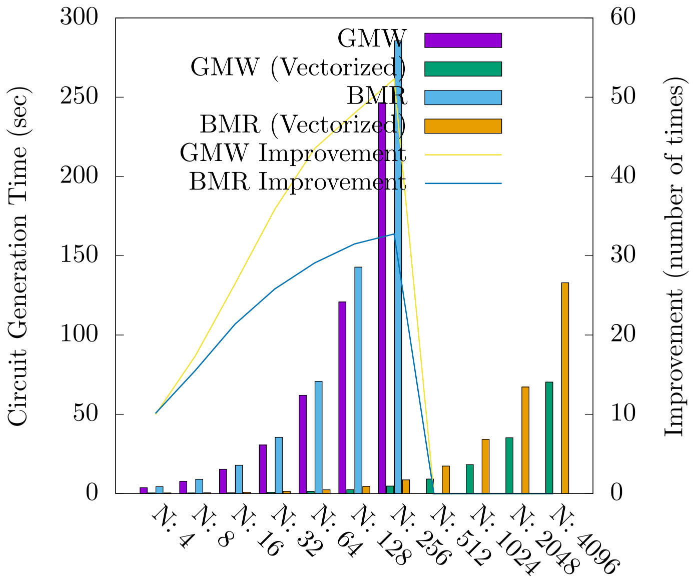

Figure 7: 2PC: Biometric Matching Circuit Gen Time, x-axis is DB-size

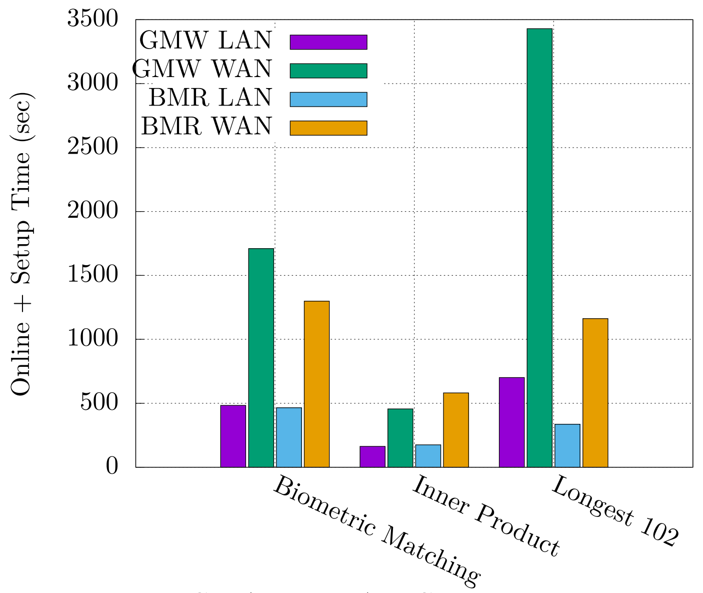

Figure 8: 2PC: LAN vs. WAN: Circuit Evaluation Time

(up to 200x less in GMW, 80x less in BMR). Online time and setup time are presented in Fig. 12, and Fig. 13 respectively.

Let us zoom into the Biometric Matching benchmark in figures 5, 6, and 7. For input size beyond `N=256` the memory usage exceeds available memory and prevents circuit generation. Therefore, non-vectorized bars are missing beyond this threshold in the graphs. Notice that vectorization improves all metrics. In circuit evaluation (see Fig. 5), BMR sees higher speedup (23x faster) compared to GMW (10x faster), while GMW sees faster circuit generation time at 45x lower (see Fig. 7) compared to BMR's which is 35x lower. Communication size reduction (see Fig. 6) is higher for GMW (10x less) compared to BMR (2.5x less).

Since our vectorization framework is network agnostic, it produces the same circuit for both LAN and WAN. Hence, the number of gates and communication size remain the same. Moreover, time for circuit generation, which is a local operation, also does not change. Setup and Online times, however, increase due to lower bandwidth and higher latency of the WAN. Indeed, this is what we observe in Fig. 8.

8 Related Work

Automatic vectorization is a longstanding problem in high-performance computing (HPC). We presented a vectorization algorithm for MPC-IR, by adapting and extending classical loop vectorization [AK87]. 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-IR vectorization. We view Karrenberg's work on Whole function vectorization [Kar15] 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). We argue that vectorization over linear MPC-IR is a problem 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 interacts with those optimizations in non-trivial ways.

Polyhedral parallelization [Ben+10] is another rich area. It considers a higher-level source (typically AST) representation, while, in contrast, our work takes advantage of linear MPC-IR and SSA form.

The early MPC compilers Fairplay [BNP08], and Sharemind [BLW08] were followed by PICCO [ZSB13], Obliv-C [ZE15], TinyGarble [Son+15], Wystiria [RHH14a], and others. A new generation of MPC compilers includes SPDZ/SCALE-MAMBA/MP-SPDZ [Kel20] and the ABY/HyCC/MOTION [DSZ15; Büs+18; Bra+22] frameworks. These two families are the state-of-the art and are actively developed. Another recent development is Viaduct, a language and compiler that supports a range of secure computation frameworks, including MPC and ZKP. Hastings et el. present a review of compiler frameworks [Has+19]. In contrast to these works, we focus on an IR and backend-independent optimizations.

The ABY/ABY3/ABY2.0/MOTION line of compiler frameworks provide excellent libraries of MPC primitives but leave it to the programmer to annotate the program properly to take advantage of available features e.g., using SIMD operations or mixing protocols. There is interest in frameworks for automatic mixing, e.g., [Büs+18; IMZ19; Fan+22].

Obliv-C [ZE15], Wysteria [RHH14b] and Viaduct [Aca+21] focus on higher-level language design. ObliVM [Liu+15] has similar goals to ours but our works are complementary in the sense that while ObliVM relies on programmer annotations such as map-reduce constructs, we automatically detect opportunities for optimization at an intermediate level of representation. Ozdemir et al. [OBW20] develop CirC, an IR with backends into zero-knowledge proof primitives as well as SMT primitives. Our work focuses on MPC, which CirC does not support yet. MPC-IR is a higher level of representation than CirC., e.g., it does not unroll loops.

HyCC [Büs+18] is a compiler from C Source into ABY circuits. It does source-to-source compilation with the goal to decompose the program into modules and then assign protocols to modules. In contrast, we focus on MPC-IR-level optimizations, specifically vectorization, although we envision future optimizations as well. HyCC, similarly to Buscher [Büs18] uses an of-the-shelf source-to-source polyhedral compiler [3] 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 [Ben+10; Kar15].

The MP-SPDZ [Kel20, Sec. 6.1] algorithm is different (and complementary) compared to our Shortest Common Supersequence (SCS) approach. The MP-SPDZ optimizer works at a lower level than our algorithm as it works with a basic block constructed by unrolling the loops. It is our understanding that (as the MP-SPDZ paper (Sect 6.3) references from [Büs+18]), there is a trade off in MP-SPDZ between the amount of loop unrolling and memory consumption. Thus, it is unclear if (and how) MP-SPDZ (and/or [Büs+18]) could fully vectorize a loop with a large iteration count. In contrast, our algorithm identifies opportunities for vectorization directly from the loops (it does not unroll), thereby avoiding the afore-mentioned trade-off. In fact, since we are working with just the loop bodies (without unrolling), the search space for Shortest Common Supersequence (SCS) may remain tractable even for an optimal vectorization by brute-force search. Working directly with the loops, however, increases complexity, which is why our algorithm might yield a slower (but still efficient) compilation. Importantly, because MP-SPDZ and our algorithm work at different levels, they are not in conflict. Both may be combined and applied. Indeed, applying our algorithm first will get rid of loops (by replacing them with SIMD instructions) and reduce search space for the MP-SPDZ algorithm, which may find further opportunities for parallelization.

References

[Aca+21] Cosku Acay et al. "Viaduct: an extensible, optimizing compiler for secure distributed programs." In: *ACM PLDI 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, June 2021, pp. 740–755.

[AK87] Randy Allen and Ken Kennedy. "Automatic Translation of Fortran Programs to Vector Form." In: *ACM Trans. Program. Lang. Syst.* 9.4 (1987), pp. 491–542.

[AN88] Alexander Aiken and Alexandru Nicolau. "Optimal Loop Parallelization." In: *ACM PLDI 1988*. Ed. by Richard L. Wexelblat. ACM, June 1988, pp. 308–317.

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

[Ara+18] Toshinori Araki et al. “Generalizing the SPDZ Compiler For Other Protocols.” In: *ACM CCS 2018*. Ed. by David Lie et al. ACM Press, Oct. 2018, pp. 880–895.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[Bea92] Donald Beaver. “Efficient Multiparty Protocols Using Circuit Randomization.” In: *CRYPTO’91*. Ed. by Joan Feigenbaum. Vol. 576. LNCS. Springer, Heidelberg, Aug. 1992, pp. 420–432.

[Ben+10] Mohamed-Walid Benabderrahmane et al. “The Polyhedral Model Is More Widely Applicable Than You Think.” In: *Compiler Construction, CC 2010*. Ed. by Rajiv Gupta. Vol. 6011. Springer, 2010, pp. 283–303.

[BG11] Marina Blanton and Paolo Gasti. “Secure and Efficient Protocols for Iris and Fingerprint Identification.” In: *ESORICS 2011*. Ed. by Vijay Atluri and Claudia Díaz. Vol. 6879. LNCS. Springer, Heidelberg, 2011, pp. 190–209.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract).” In: *20th ACM STOC*. ACM Press, May 1988, pp. 1–10.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. “Sharemind: A Framework for Fast Privacy-Preserving Computations.” In: *ESORICS 2008*. Ed. by Sushil Jajodia and Javier López. Vol. 5283. LNCS. Springer, Heidelberg, Oct. 2008, pp. 192–206.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. “The Round Complexity of Secure Protocols (Extended Abstract).” In: *22nd ACM STOC*. ACM Press, May 1990, pp. 503–513. DOI: `10.1145/100216.100287`.

[BNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. “FairplayMP: a system for secure multi-party computation.” In: *ACM CCS 2008*. Ed. by Peng Ning, Paul F. Syverson, and Somesh Jha. ACM Press, Oct. 2008, pp. 257–266.

[Bog+09] Peter Bogetoft et al. “Secure Multiparty Computation Goes Live.” In: *FC 2009*. Ed. by Roger Dingledine and Philippe Golle. Vol. 5628. LNCS. Springer, Heidelberg, Feb. 2009, pp. 325–343.

[Bra+22] Lennart Braun et al. “MOTION: A Framework for Mixed-Protocol Multi-Party Computation.” In: *ACM TOPS* 25.2 (May 2022), pp. 1–35.

[Büs+18] Niklas Büscher et al. “HyCC: Compilation of Hybrid Protocols for Practical Secure Computation.” In: *ACM CCS 2018*. Ed. by David Lie et al. ACM Press, Oct. 2018, pp. 847–861.

[Büs18] Niklas Büscher. “Compilation for More Practical Secure Multi-Party Computation.” PhD thesis. Darmstadt University of Technology, Germany, 2018.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. “Multiparty Unconditionally Secure Protocols (Extended Abstract).” In: *20th ACM STOC*. ACM Press, May 1988, pp. 11–19.

[Cyt+91] Ron Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.” In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), 451?–490. ISSN: 0164-0925.

[Dow+16] Nathan Dowlin et al. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy.” In: *ICML 2016*. New York, NY, USA: JMLR.org, June 2016, pp. 201–210.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. “ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation.” In: *NDSS 2015*. The Internet Society, Feb. 2015.

[Dup+19] Dmitry Duplyakin et al. “The Design and Operation of CloudLab.” In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14.

[Esc+20] Daniel Escudero et al. “Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits.” In: *CRYPTO 2020, Part II*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. LNCS. Springer, Heidelberg, Aug. 2020, pp. 823–852.

[Fan+22] Vivian Fang et al. *CostCO: An automatic cost modeling framework for secure multi-party computation.* Cryptology ePrint Archive, Report 2022/332. `https://eprint.iacr.org/2022/332`. 2022.

[FN21] Azadeh Farzan and Victor Nicolet. "Phased synthesis of divide and conquer programs." In: *ACM PLDI 2021.* Ed. by Stephen N. Freund and Eran Yahav. ACM, July 2021, pp. 974–986.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority." In: *19th ACM STOC.* Ed. by Alfred Aho. ACM Press, May 1987, pp. 218–229.

[Has+19] Marcella Hastings et al. "SoK: General Purpose Compilers for Secure Multi-Party Computation." In: *2019 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 2019, pp. 1220–1237.

[IMZ19] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. "Efficient MPC via Program Analysis: A Framework for Efficient Optimal Mixing." In: *ACM CCS 2019.* Ed. by Lorenzo Cavallaro et al. ACM Press, Nov. 2019, pp. 1539–1556.

[JW05] Geetha Jagannathan and Rebecca N. Wright. "Privacy-Preserving Distributed k-Means Clustering over Arbitrarily Partitioned Data." In: *ACM CKDDM.* Chicago, IL, USA: ACM, 2005, pp. 593–599.

[Kar15] Ralf Karrenberg. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs.* Springer, 2015. ISBN: 978-3-658-10112-1.

[Kel20] Marcel Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation." In: *ACM CCS 2020.* Ed. by Jay Ligatti et al. ACM Press, Nov. 2020, pp. 1575–1590.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer." In: *ACM CCS 2016.* Ed. by Edgar R. Weippl et al. ACM Press, Oct. 2016, pp. 830–842.

[Liu+15] Chang Liu et al. "ObliVM: A Programming Framework for Secure Computation." In: *2015 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 2015, pp. 359–376.

[MB08] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver." In: *TACAS 2008.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Springer, Apr. 2008, pp. 337–340.

[MR18] Payman Mohassel and Peter Rindal. "ABY^3: A Mixed Protocol Framework for Machine Learning." In: *ACM CCS 2018.* Ed. by David Lie et al. ACM Press, Oct. 2018, pp. 35–52.

[MZ17] Payman Mohassel and Yupeng Zhang. "SecureML: A System for Scalable Privacy-Preserving Machine Learning." In: *2017 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 2017, pp. 19–38.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL.* Heidelberg, Germany: Springer, 2014. ISBN: 3319105418.

[OBW20] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. *Unifying Compilers for SNARKs, SMT, and More.* Cryptology ePrint Archive, Report 2020/1586. `https://eprint.iacr.org/2020/1586`. 2020.

[Pat+21] Arpita Patra et al. "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation." In: *USENIX Security 2021.* Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 2021, pp. 2165–2182.

[RHH14a] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations." In: *2014 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 2014, pp. 655–670.

[RHH14b] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. “Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations.” In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* 2014, pp. 655–670.

[Sco09] Michael L. Scott. *Programming Language Pragmatics (3. ed.)* Academic Press, 2009.

[Son+15] Ebrahim M. Songhori et al. “TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits.” In: *2015 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 2015, pp. 411–428.

[Vaz10] Vijay V. Vazirani. *Approximation Algorithms.* Heidelberg, Germany: Springer, 2010. ISBN: 3642084699.

[VC03] Jaideep Vaidya and Chris Clifton. “Privacy-Preserving k-Means Clustering over Vertically Partitioned Data.” In: Washington, D.C.: ACM, 2003, pp. 206–215.

[Yao82] Andrew Chi-Chih Yao. “Protocols for Secure Computations (Extended Abstract).” In: *23rd FOCS.* IEEE Computer Society Press, Nov. 1982, pp. 160–164.

[ZE15] Samee Zahur and David Evans. *Obliv-C: A Language for Extensible Data-Oblivious Computation.* Cryptology ePrint Archive, Report 2015/1153. 2015.

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. “PICCO: a general-purpose compiler for private distributed computation.” In: *ACM CCS 2013.* Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. ACM Press, Nov. 2013, pp. 813–826.

appendix

9 Proofs

9.1 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 $\{s_1, \dots, s_n\}$ 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. $M_1 \to M_2 \to M_1$
2. $M_1 \to M_1 \to M_1$
3. $M_2 \to M_1 \to M_2$

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

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

The parentheses above indicate the sequence where the instruction comes from: (1), (2), or (3). Cost of schedule P is computed using Eq. (2) 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 [Vaz10]. 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. $\square$

9.2 Proof of Theorem 1

Before proving the theorem, it helps to prove the following lemma, which states that Basic Vectorization preserves statements and def-use edges in the original MPC-IR.

Lemma 1. *For each statement s in a_0, there is same statement s' in a_1, and vice versa. For each def-use edge e in a_0, there is a same edge e' in a_1, and vice versa.*

Proof. Proof sketch of Lemma 1. Phase 2 of Basic Vectorization does not introduce any new statements in the code, it just vectorizes dimensions. Similarly, reordering of statements preserves exactly the def-use edges in the original MPC-IR. $\square$

Proof of the theorem follows:

Proof. The first condition of property P follows directly from Lemma 1. The proof of the second condition is by analysis of the def-use edges in $\gamma(a_0)$ and the corresponding edges in $\gamma(a_1)$; as mentioned earlier, the key is that Basic Vectorization preserves the def-uses in a_0.

A forward edge $s_0 \to s_1 \in a_0$ remains a forward edge in a_1. Without loss of generality, let us assume an outer loop i and a nested loop j. The forward edge entails the following ordering in linearization $\gamma(a_0)$:

$\underline{s_0}[\underline{i}]$; $\underline{s_1}[\underline{i},\underline{j}]$	outer-to-inner edge
$\underline{s_0}[\underline{i},\underline{j}]$; $\underline{s_1}[\underline{i},\underline{j}]$	same-level edge
$\underline{s_0}[\underline{i},\underline{j}]$; $\underline{s_1}[\underline{i}]$	inner-to-outer edge

meaning that for a fixed $\underline{i}$, def $\underline{s_0}[\underline{i}]$ is scheduled *before* use $\underline{s_1}[\underline{i}]$. Due to the preservation of the edge in a_1, the above ordering holds in $\gamma(a_1)$ as well.

Consider a backward edge $s_0 \to s_1 \in a_0$. We have that s_1 is a PHI-node in some loop, say i. There are two cases: (1) there is a path of forward edges from s_1 to s_0, and (2) there is no such path. In case (1), Basic Vectorization detects a cycle (closure) around s_1, and therefore, $s_0 \to s_1$ remains a backward edge in a_1. The linearization of the backward edge imposes ordering $\underline{s_0}[\underline{i}-1]$; $\underline{s_1}[\underline{i}]$ and due to preservation of the backward edge in a_1, the ordering holds in $\gamma(a_1)$ as well. In case (2), Basic Vectorization my turn the backward edge into a forward one, however, it preserves the $\underline{s_0}[\underline{i}-1]$; $\underline{s_1}[\underline{i}]$ ordering constraint by construction.

$\square$

9.3 Proof of Corollary 1.1

Proof. Proof sketch of Corollary 1.1. This can be established by induction over the length of def-use chains of computation in $\gamma(a_0)$. Assume that for all chains of length $\leq n$ all locations $l[\underline{i},\underline{j},\underline{k}]$ hold the same value in $\gamma(a_0)$ and $\gamma(a_1)$. A chain of length $n+1$ results from the execution of a statement $\mathsf{x}[\underline{i},\underline{j},\underline{k}] = \mathsf{y}[\underline{i},\underline{j},\underline{k}]$ *op* $\mathsf{z}[\underline{i},\underline{j},\underline{k}]$. By property P, there is the same statement in $\gamma(a_1)$ and it is scheduled after the definitions of $\mathsf{y}[\underline{i},\underline{j},\underline{k}]$ and $\mathsf{z}[\underline{i},\underline{j},\underline{k}]$. By the inductive hypothesis $\mathsf{y}[\underline{i},\underline{j},\underline{k}]$ and $\mathsf{z}[\underline{i},\underline{j},\underline{k}]$ hold the same values in $\gamma(a_0)$ as in $\gamma(a_1)$. Therefore, locations $\mathsf{x}[\underline{i},\underline{j},\underline{k}]$ hold the same value as well. We remark that due to the SSA form, each location $l[\underline{i},\underline{j},\underline{k}]$ is defined at most once. For clarity, we elide PHI nodes and raising and dropping dimensions; extending def-use reasoning is straight forward. $\square$

10 Compiler Frontend

10.1 From IMP Source to E-SSA

Our compiler translates from Source to E-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 §4.2. This step outputs an instance of the `restricted_ast.Function` class, which represents our restricted subset of the Python AST.

3-address E-CFG conversion: Convert the restricted-syntax AST to a three-address enhanced 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 §4.2). Rules for processing each kind of statement are given below:

- **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.
- **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.
- **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., $\mathsf{A[i]} = \mathsf{x}$, change the statement to $\mathsf{A} = \mathsf{Update(A, i, x)}$. 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.

10.2 From SSA to MPC-IR

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 $\mathsf{X!3} = \phi\mathsf{(X!1,X!2)}$ is in a block with merge condition C, the compiler transforms it into $\mathsf{X!3} = \mathsf{MUX(C, X!1, X!2)}$. Next, the compiler runs the dead code elimination algorithm from Cytron's SSA paper.

Next, the control-flow graph is *linearized* into MPC-IR, 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 depth-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-IR. If the block ends in a for-instruction, the compiler recursively converts the body and code after the loop to MPC-IR and adds the for-loop and code after the loop to the main MPC-IR. If the block does not end in a for-instruction, the compiler recursively converts all successor branches to MPC-IR and appends these to the main MPC-IR.

```
{ Step 1: Replace $\phi$-nodes with MUX nodes }
for each basic block $block$ in the control-flow graph do
  if $block$ has the merge condition property then
    $merge\_cond$ ← merge condition variable of $block$
    for each $\phi$-node $phi = \phi(v_1, v_2)$ in $block$ do
      Replace $phi$ with MUX($merge\_cond, v_1, v_2$) in $block$
    end for
  end if
end for
{ Step 2: Linearize E-CFG into MPC-IR }
$visited$ ← empty set
$merge\_visited$ ← empty set
Define search($block$):
  if $block$ has the merge condition property and $block$ is not in $merge\_visited$ then
    Add $block$ to $merge\_visited$
    return empty list
  end if
  Add $block$ to $visited$
  if $block$ is a for-loop header then
    $cfg\_body$ ← successor of $block$ containing the beginning of the for-loop body
    $cfg\_after$ ← successor of $block$ containing the beginning of the code after the for-loop
    $mpc\_body$ ← list of the $\phi$-functions in $block$ concatenated with search($cfg\_body$)
    $loop$ ← MPC-IR for-loop statement with the same counter variable and bounds as $block$ and with
    $mpc\_body$ as its body
    return $loop$ prepended to search($cfg\_after$)
  else
    $result$ ← empty list
    for each successor $successor$ of $block$ do
      if $successor$ is not in $visited$ then
        $result$ ← $result$ concatenated with search($successor$)
      end if
    end for
    return $result$
  end if
return search(entry block of the control-flow graph)
```

Now, the remaining ϕ-nodes in MPC-IR are the loop header nodes. These are the *pseudo* ϕ-nodes and we write PHI in MPC-IR. A pseudo PHI-node x!1 = PHI(x!0,x!2) in a loop header is evaluated during circuit generation. If it is the 0-th iteration, then the PHI-node evaluates to x!0, otherwise, it evaluates to x!2.

10.3 Base MPC-IR Syntax and Taint Types

The syntax of the MPC-IR program produced by the above section is essentially IMP syntax. (In section 5 we extended the base syntax to account for vectorization.) Most notably, there is no if-then-else statement but there are MUX expressions:

$$\begin{array}{ll} e ::= e \; op \; e \mid \mathsf{x} \mid \texttt{const} \mid \texttt{A[}e\texttt{]} \mid \texttt{MUX}(e,e,e) & \textit{expression} \\ s ::= s; s \mid & \textit{sequence} \\ \quad \mathsf{x} = \texttt{PHI}(\mathsf{x},\mathsf{x}) \mid \mathsf{x} = e \mid \texttt{A[}e\texttt{]} = e \mid & \textit{assignment stmt} \\ \quad \texttt{for } i \texttt{ in range(}I\texttt{) : } s & \textit{for stmt} \end{array}$$

Expressions are typed $\langle q\ \tau \rangle$, where q and τ are as follows:

$$\begin{array}{ll} \tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{list[int]} \mid \texttt{list[bool]} & \textit{base types} \\ q ::= \texttt{shared} \mid \texttt{plain} & \textit{qualifiers} \end{array}$$

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

The typing rules for non-trivial expressions are as follows:

$$\frac{\Gamma \vdash e_1 : \langle q_1\ \tau \rangle \quad \Gamma \vdash e_2 : \langle q_2\ \tau \rangle \quad \tau \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash e_1\ op\ e_2 : \langle q_1 \vee q_2\ \tau \rangle}\ \text{(Binary Op)}$$

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

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

Similarly, the typing rules for statements are as follows. The constraints are standard: the right-hand side of an assignment is a subtype of the left-hand side.

$$\frac{\Gamma \vdash \mathsf{x}_1 : \langle q_1\ \tau \rangle \quad \Gamma \vdash \mathsf{x}_2 : \langle q_2\ \tau \rangle \quad \Gamma \vdash \mathsf{x}_3 : \langle q_1\ \tau \rangle \quad q_2 \vee q_3 <: q_1}{\Gamma \vdash \mathsf{x}_1 = \texttt{PHI}(\mathsf{x}_2, \mathsf{x}_3) : OK}\ \text{(PHI Assign)}$$

$$\frac{\Gamma \vdash \mathsf{x} : \langle q_1\ \tau \rangle \quad \Gamma \vdash e : \langle q_2\ \tau \rangle \quad q_2 <: q_1 \quad \tau \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash \mathsf{x} = e : OK}\ \text{(Var Assign)}$$

$$\frac{\Gamma \vdash i : \langle \texttt{plain int} \rangle \quad \Gamma \vdash I : \langle \texttt{plain int} \rangle \quad \Gamma \vdash s : OK}{\Gamma \vdash \texttt{for } i \texttt{ in range(}I\texttt{) : } s : OK}\ \text{(For Stmt)}$$

As mentioned earlier, the only annotations the program need provide is on program inputs. The compiler infers the rest of the annotations. The type system has two purposes (1) it imposes restrictions, and (2) it enables code generation, specifically, it informs the backend on weather a statement operates on shared variables or plaintext ones, and the backend generates appropriate MOTION code.

11 Backend-Independant Vectorization

This appendix contains the additional content corresponding to §5 that had to be cut from main body of the paper due to page limit.

The algorithm for basic vectorization is listed and described in §5.3, here we explain it further via its application to the running example of the paper, Biometric Distance.

11.1 Example: Biometric

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

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

```
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))
for i in range(0, N):
    min_sum!2 = PHI(min_sum!1^, min_sum!4)
    min_idx!2 = PHI(min_idx!1^, min_idx!4)
    sum!2 = 0 // Will lift, when hoisted
    sum!2^ = raise_dim(sum!2, (j:D))
    for j in range(0, D):
        sum!3 = PHI(sum!2^, sum!4)
        d = SUB(S^,C^)
        p = MUL(d,d)
        sum!4 = ADD(sum!3,p)
    sum!3^ = drop_dim(sum!3)
    t = CMP(sum!3^,min_sum!2)
    min_sum!3 = sum!3^
    min_idx!3 = i // Same−level, will lift when hoisted
    min_sum!4 = MUX(t, min_sum!3, min_sum!2)
    min_idx!4 = MUX(t, min_idx!3, min_idx!2)
min_sum!2^ = drop_dim(min_sum!2)
min_idx!2^ = drop_dim(min_idx!2)
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 re-introduce 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)
```

11.2 Examples with Array Writes

In this section, we present several examples to demonstrate the vectorization algorithm on 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:

```
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]);
```

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-IR 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
```

12 Compiler Back End

MOTION framework requires that all variables are marked as `plain` or `shared` following the type system in §4.2. We require that only inputs are marked as either shared or plaintext, and infer qualifiers for other variables through *taint analysis* of §12.1. We provide details of code generation for MOTION backend in §12.2.

12.1 Taint Analysis

The taint analysis works on MPC-IR, 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.

12.2 From (Optimized) MPC-IR to MOTION

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

```
A[i] = val
```

IMP Source

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

MPC-IR

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

MOTION Code

Table 3: MOTION Translation: Array Updates

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

MPC-IR

```
_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 4: MOTION Translation: FOR loop with Phi nodes

Variable declarations: Our generated C++ uses the following variable-naming scheme: shared variables are named the same as in the MPC-IR 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-IR is translated into C++ one statement at a time. The linear structure of MPC-IR 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 3.

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 4 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

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

MPC-IR

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

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

MPC-IR

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

MOTION Code

Table 6: MOTION Translation: Raising dimensions

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 5 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 §5.2. 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 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.

13 Evaluation

For the 3PC, we observe that, as expected, evaluation time (Fig. 14) is higher than the 2PC. This is a direct consequence of higher online time (Fig. 15) for the GMW protocol. Online time for BMR remains roughly the same, which is expected because online phase is essentially local in BMR. BMR suffers a slow down in the setup phase (Fig. 16) however. This is due to the circuit for 3 parties requiring more computation. Due to space restriction, we do not include graphs for (1) circuit generation time, (2) gates count, and (3) communication size for 3PC. Circuit generation sees a slow down in BMR for the reason mentioned above, communication size per channel and gates count remain the same. The experiments for 3PC essentially provide confirmation that adding more parties to an MPC increases resource requirement.

We also leave out detailed graphs (similar to the ones we include for Biometric Matching in the main body of this paper) for Inner Product.

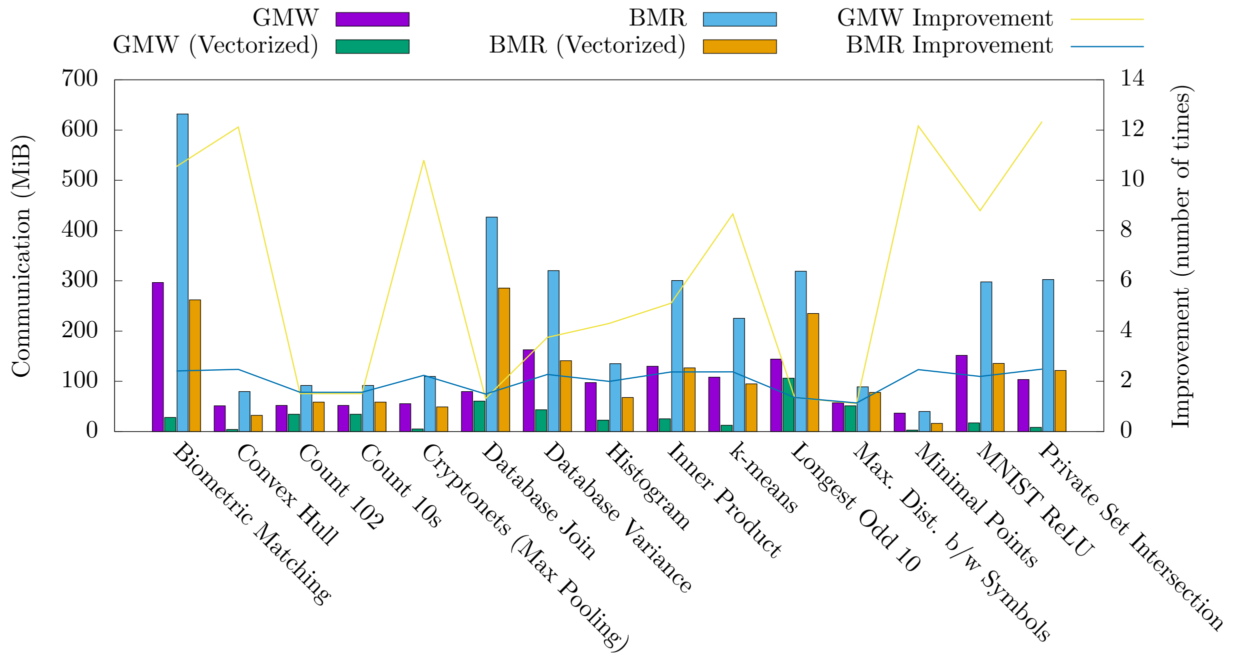

Figure 9: 2PC: Communication Size of Benchmarks

13.1 Analysis: Communication Size Reduction

As shown in communication size graph (Fig. 9), vectorization results in reduced communication (fewer bits are transferred). This reduction is a result of more efficient data-packing at both (1) the application level (i.e. the MPC backend level), and (2) at the network level. The MPC backend needs to store/send metadata with each primitive/message so that it can correctly decoded/consumed later. For example, a gate needs an identifier *gid*, a gate type *gtype*, incoming wire identifiers, etc. Say *size*(*gmeta*), bits are needed to

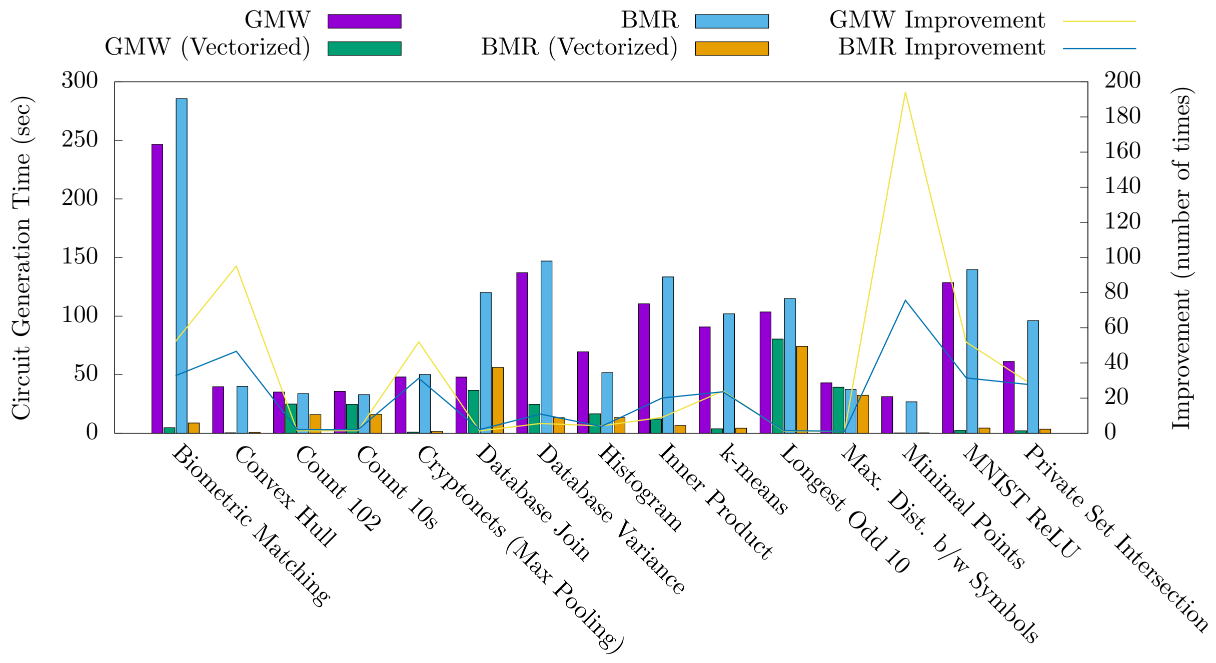

Figure 10: 2PC: Circuit Generation Time of Benchmarks

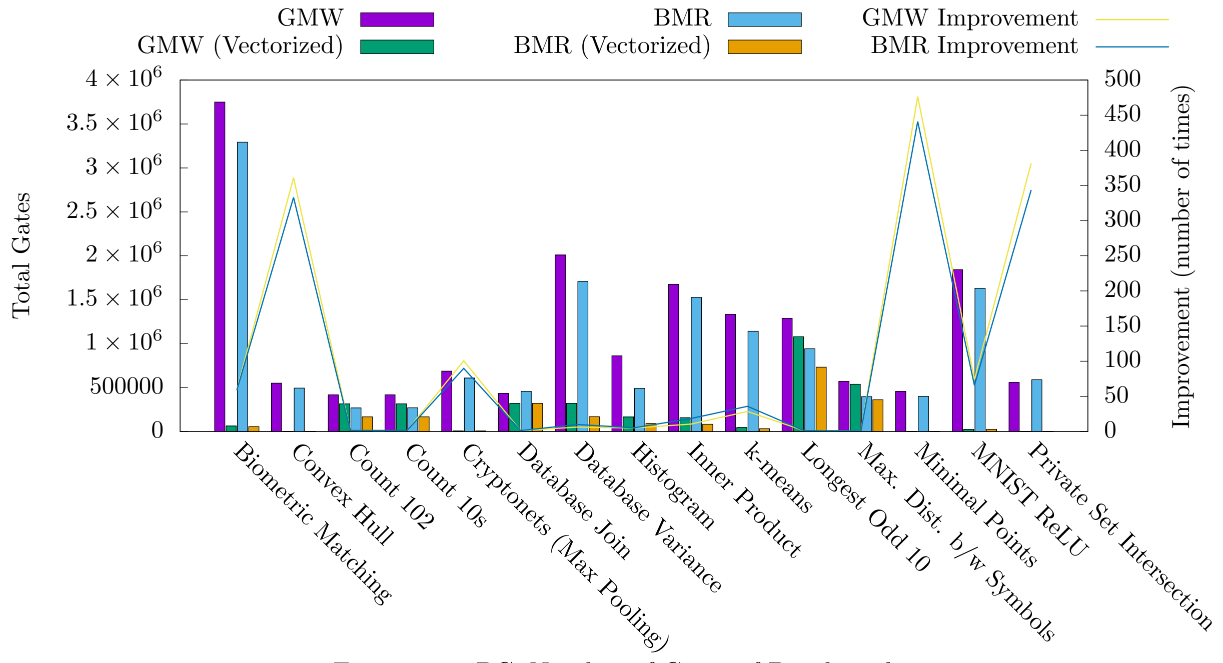

Figure 11: 2PC: Number of Gates of Benchmarks

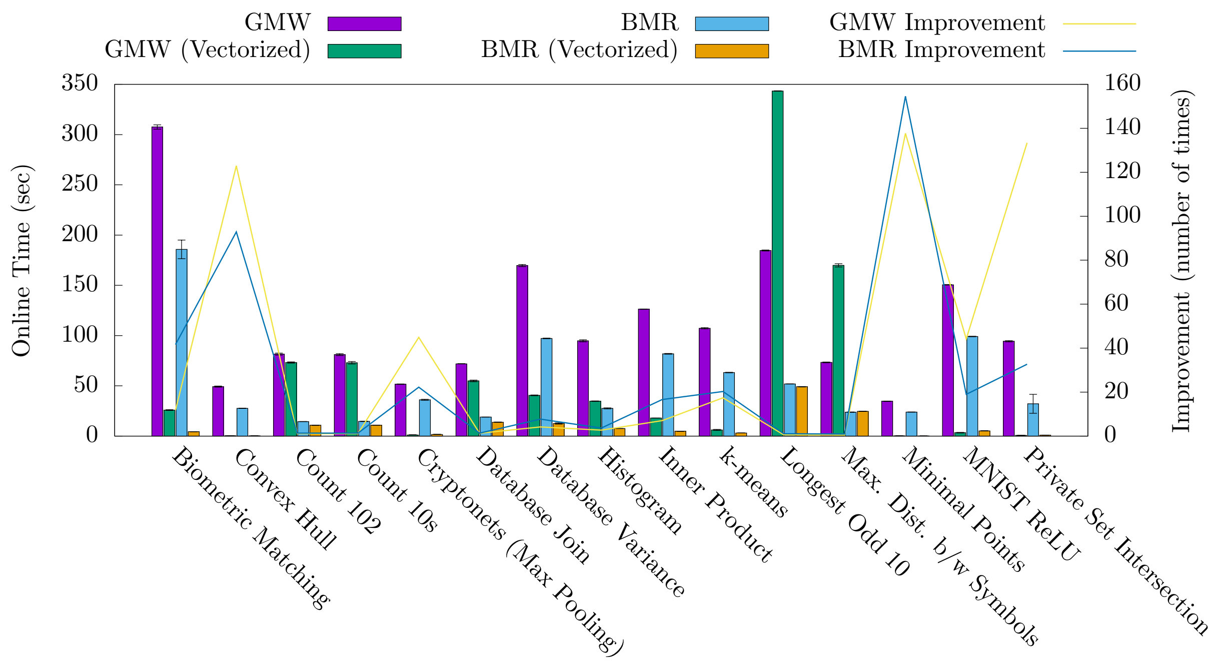

Figure 12: 2PC: Online Time of Benchmarks

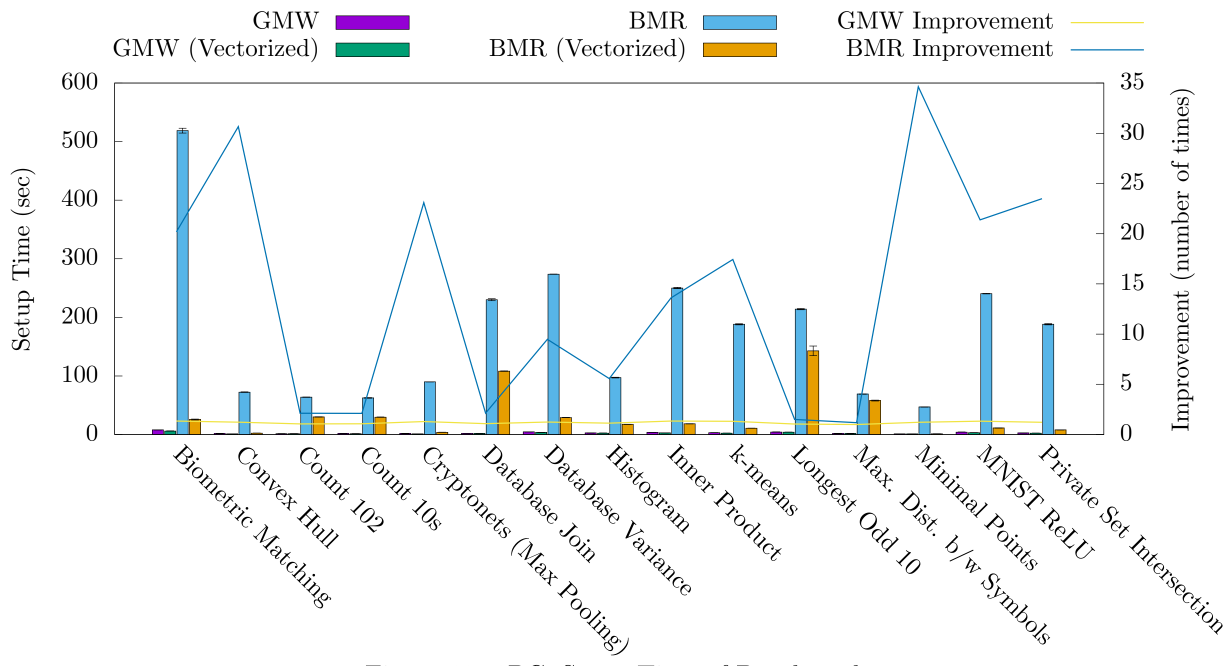

Figure 13: 2PC: Setup Time of Benchmarks

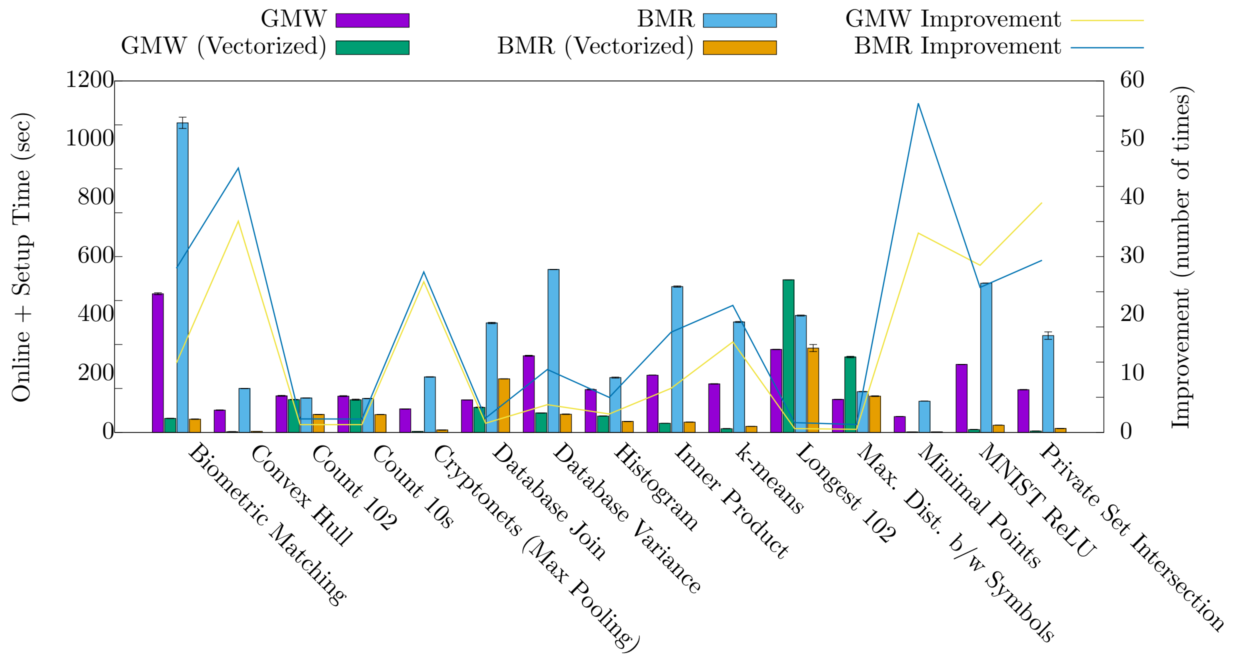

Figure 14: 3PC: Circuit Evaluation Time (Setup and Online Phase) in Seconds, LAN Setting. The Error-bars are Standard Deviation.

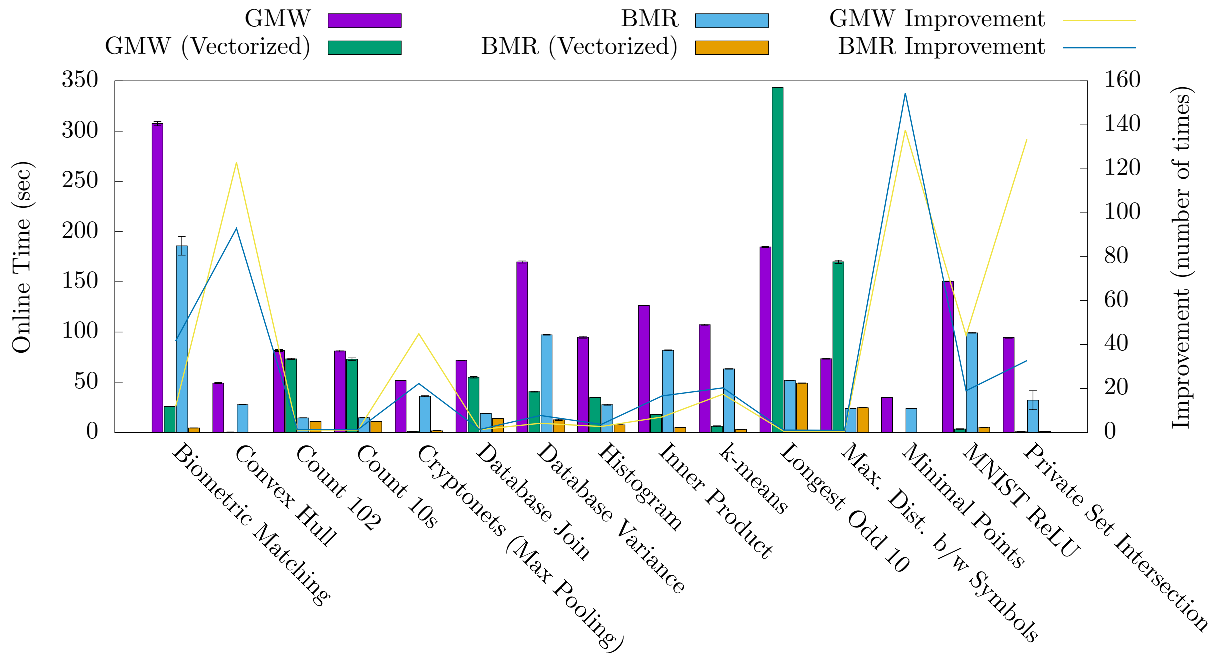

Figure 15: 3PC: Online Time (in Seconds) of Benchmarks, LAN Setting. The Error-bars are Standard Deviation.

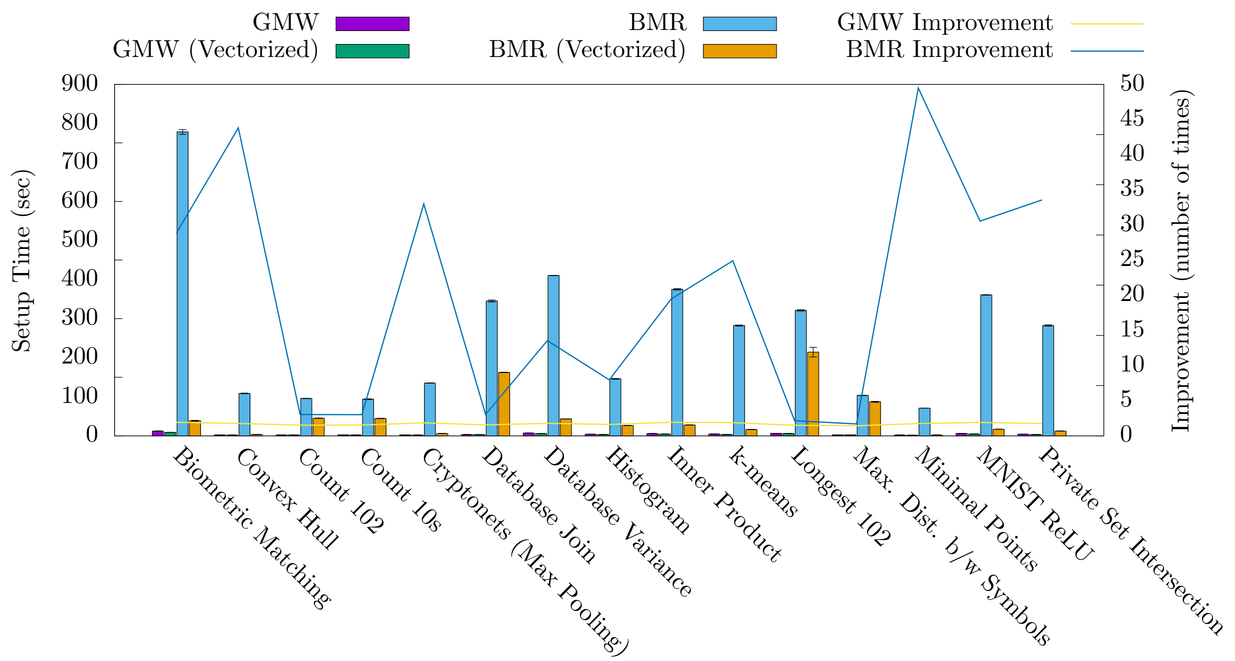

Figure 16: 3PC: Setup Time (in Seconds) of Benchmarks, LAN Setting. The Error-bars are Standard Deviation.

store/send metadata for a single gate. Using one vectorized/SIMD gate instead of $(N + 1)$ non-vectorized gates saves $N \cdot size(gmeta)$ bits in memory/communication. Similarly, at the network level, each message needs a header h that contains routing and decoding information while the packet is in transit. Say, one (non-vectorized) interactive gate induces a payload p. This means, $size(h) + size(p)$ bits are sent to network for each (non-vectorized) interactive gate. Evaluation of N such gates translates to $N \cdot (size(h) + size(p))$ bits of communication. On the other hand, a vectorized gate that replaces these N gates is much cheaper, and requires only $size(h) + N \cdot size(p)$ bits of communication.

Concretely, let us consider an MPC backend implemented on Transport Control Protocol (TCP) over Internet Protocol (IP) i.e., most common communication stack. Note that, for the sake of communication size comparison, the only difference between UDP and TCP is the smaller header size of 8 bytes[4] in UDP compared to the at least 20 bytes[5] in TCP. Both protocols are typtically implemented over Internet Protocol and the header size of an IPv4 packet is 20 bytes[6]. In the Arithmetic GMW protocol, multiplication operation (MUL) is typically implemented using Beaver's triples [Bea92]. This means that, in the online phase, all parties need to send 2ℓ bits to each other. If $\ell = 32$ bits, then, TCP payload is $2\ell = 64$ bits or 8 bytes. Considering that the Maximum Transmission Unit (MTU) is typically 1500 bytes, a TCP message may have payload of up to $(1500 - 20 - 20) = 1460$ bytes (the exact value is decided via the Maximum Segment Size (MSS) during TCP stack initialization, specification maximum is 65,496 bytes). Meaning, a vectorized gate replacing $1460/8 \approx 180$ non-vectorized gates could be sent in a single 1500 byte message rather than $182 \cdot (20 + 20 + 8) = 8,640$ bytes otherwise required. Similar reasoning applies to interactive gates in boolean GMW and, while exact improvement depends on the implementation details, packing data reduces both the memory and communication footprint regardless of the underlying MPC backend (as long as it supports vectorized gates).

In the case of BMR, the entire circuit can be packed as one payload and sent using a few TCP packets. Therefore under-utilization of network's payload-capacity is not an issue. At the application (MPC backend)

[4]`https://en.wikipedia.org/wiki/User_Datagram_Protocol\#UDP_datagram_structure`
[5]`https://en.wikipedia.org/wiki/Transmission_Control_Protocol\#TCP_segment_structure`
[6]`https://en.wikipedia.org/wiki/IPv4\#Header`

level however, inefficient packing is still a problem. For example, MOTION uses 64 bits (8 bytes) for gate identifiers. A vectorized gate that replaces 128 non-vectorized gates, requires only one gate identifier i.e. 8 bytes instead 1,024 bytes required for 128 identifiers. Thus, vectorization reduces the size of the circuit. This, in turn, reduces payload for the network and means that fewer TCP packets need to be sent, thereby saving on TCP/IP metadata that would have been needed for additional packets.

13.2 Comparison with MOTION-native Inner Product

When comparing our results with the manually SIMD-ified ones distributed with MOTION source, we noticed a peculiarity in the case of Inner Product. 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 turns out that the vectorized multiplication are the same, however, our addition loop incurs significant cost (ADD is non-local and expensive in Boolean GMW). The MOTION-native loop runs

```
result += mult_unsimdified[i];
```

while our loop generates and runs

```
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 lead to the same running time.

MOTION's compiler performs analysis that informs circuit generation and the example illustrates the power of the analysis. In the above example, MOTION overloads the += operator to perform divide-and-conquer accumulation in $O(log(N))$ rounds. Recall that the next phase of Vectorization, in §5, 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 code. It still might lead to confusion in the static analysis as analysis on the AST is difficult. Our investigation showed that not only MOTION does not optimize

```
if (i!=0) {
  result_prev = result;
}
result = result_prev + mult_unsimdified[i];
```

it does not optimize the simpler accumulation:

```
result = result + mult_unsimdified[i];
```

We conjecture that MPC-IR, 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 [FN21]; as the problem is non-trivial, particularly the interaction of divide-and-conquer with vectorization and mixing, we leave it for future work.