Abstract

Recent years have witnessed significant advances in programming technology for multi-party computation (MPC), bringing MPC closer to practice and wider applicability. Typical MPC programming frameworks focus on either front-end language design (e.g., Wysteria, Viaduct, SPDZ), or back-end protocol implementation (e.g., ABY, MOTION, SPDZ).

We propose a methodology for an MPC compilation toolchain, which by mimicking the compilation methodology of classical compilers enables middle-end (i.e., machine-independent) optimizations, yielding significant improvements. We advance an intermediate language, which we call MPC-IR that can be viewed as the analogue of (enriched) Static Single Assignment (SSA) form. MPC-IR enables backend-independent optimizations in a close analogy to machine-independent optimizations in classical compilers. To demonstrate 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. To demonstrate backend independence and quality of our optimization, we evaluate our approach with two mainstream backend frameworks that support multiple types of MPC protocols, namely MOTION and MP-SPDZ, and show significant improvements across the board.

Contents

1 Introduction 2
   1.1 Our Contribution .......................................................... 3

2 Overview of our Methodology 5

3 Related Work 6
   3.1 MPC Compilers ............................................................... 6
   3.2 Intermediate representations for MPC ................................. 7
   3.3 Automatic vectorization in HPC ........................................... 7

*Work done while the author was a graduate student at RPI.
†Work done while the author was a graduate student at RPI.
## 1 Introduction

Multi-party computation (MPC) allows $N$ parties $P_1, \ldots, P_N$ to perform a computation on their private inputs securely. Informally, security means that the secure computation protocol computes the correct output (correctness) and it does not leak any information about the individual party inputs 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 goal is to enable 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], Wys* [RSH19], 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; RSH19; Aca+21]. The latter, backend-focused thread devised cryptographic protocols, typically at the circuit-level [DSZ15; Ara+18; Bra+22; Pat+21; KOS16].

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. We envision different front ends compiling into MPC-IR — the frontend we use in this work is IMP Source, an easy to use Python-like language.

MPC-IR exposes the linear structure of MPC programs, which simplifies program analysis; this is in contrast to IMP source, which has branching constructs. At 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; this makes it suitable for defining optimizations that take advantage of amortization at the circuit level such as SIMD-vectorization and protocol mixing [Büs+18; IMZ19; Fan+22; Esc+20; Che+23]. Importantly, our formalization enables reasoning about optimizations over MPC-IR and correctness proofs of the optimizations and transformations. We demonstrate the utility of the IR and optimizations on two MPC backends, MOTION [Bra+22] and MP-SPDZ [Kel20]. We include a more detailed exposition and comparison with related work in §3.

1.1 Our Contribution

We describe an end-to-end compiler framework that takes a Python-like routine (frontend) and produces optimized backend code. More concretely we present (a) our (Python-like) IMP Source language, its syntax and semantic restrictions; (b) formal specification of MPC-IR; (c) a translation of IMP Source into MPC-IR, (d) a specific backend-independent optimization: novel SIMD-vectorization on MPC-IR; and (e) translation of MPC-IR into MOTION and MP-SPDZ code for a large set of benchmarks.
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 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 [IMZ19; Fan+22; Che+23]. Given the restrictions on MPC programs that are imposed by privacy requirements—e.g., the linear program structure and bounded loops—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 an interesting problem. In this work, we provide an optimization by utilizing our cost model and IR and taking inspiration from the classical area of 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 $N$.

In more detail, 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 backend MPC framework supports SIMD gates—most mainstream backends do, in particular the ones chosen here—this results in smaller circuits and reduces both the time and memory footprint of circuit generation. We note in passing that although much of the applied MPC literature has focused on run-time improvements, for the technology to become mainstream it is imperative to also improve the compilation time. Our work demonstrates improvements in both.

We demonstrate the wide applicability of our framework (and the expressivity of the source language) by running the compiler on 15 programs with interleaved if- and for-statements. We further demonstrate the utility and the backend-independent nature of our optimization by generating iterative and vectorized code for two well-known backend frameworks, MOTION [Bra+22] and MP-SPDZ [Kel20], taking advantage of the respective SIMD API in vectorized code; each backend then translates the code into circuits and runs the circuits. We chose these two frameworks as they are not only among the more prominent in the relevant literature, but they are also highly parameterizable as they support several MPC paradigms. They can generate and evaluate Boolean or Arithmetic circuits with GMW-style protocols [GMW87] or Boolean circuits with BMR-style protocols [BMR90]—for brevity we refer to the corresponding protocols as (Boolean/Binary or Arithmetic) GMW and BMR. This not only allows us to showcase our optimization in a wide range of benchmarks—across different MPC paradigms (see below)—but also opens our methodology to what we view as a major application area for our optimizer, namely MPC mixing.

In more detail, for MOTION we generate code for Boolean GMW and BMR—we do not include Arithmetic GMW, as MOTION does not support all operations in the Arithmetic GMW protocol, e.g., MUX, and we do not yet support protocol mixing. Towards evaluation we run experiments with two parties (2PC) and three parties (3PC). Circuit evaluation time for vectorized code improves over iterative code up to 30x for 2PC (resp. 40x for 3PC setting) in GMW and up to 45x for 2PC (resp. 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.

For MP-SPDZ we generate Python code then compile and execute in the Arithmetic setting and in the Binary setting. We run experiments with 2PC and demonstrate reduction in compilation time of up to 25x in the Binary setting.

Full source code, benchmarks, and results on small inputs are available at https://github.com/milana2/ParallelizationForMPC where https://github.com/milana2/ParallelizationForMPC/tree/gh-pages shows experiments on small input sizes and details the different stages of the compiler. The current implementation supports unsigned integers, booleans, tuples, and lists, although it is our intention to extend it with additional types. The MPC protocol parameters e.g., computational/statistical security parameters, are specified independently for each backend. In our evaluation, we use each backend’s default values for them.
def biometric(C: shared[list[int]], D: int, N: int) -> tuple[list[int]]:
    min_sum : int = MAX_JNT
    min_idx : int = 0
    for i in range(N):
        sum : int = 0
        for j in range(D):
            d : int = S[i, j] - C[j]
            p : int = d + d
            sum = sum + p
        if sum < min_sum:
            min_sum : int = sum
            min_idx : int = i
    return (min_sum, min_idx)

min_sum1 = MAX_JNT
min_idx1 = 0
for i in range(0, N):
    min_sum2 = PHI(min_sum1, min_sum1)
    min_idx2 = PHI(min_idx1, min_idx1)
    sum2 = 0
    for j in range(0, D):
        sum3 = PHI(sum2, sum2)
        d = SUB(S[i, i + D] + j, C[j])
        p = MUL(d, d)
        sum4 = ADD(sum3, p)
        t = CMP(sum4, min_sum2)
        min_sum3 = sum4
        min_idx3 = i
        min_sum4 = MUX(t, min_sum3, min_sum2)
        min_idx4 = MUX(t, min_idx3, min_idx2)
    return (min_sum2, min_idx2)

min_sum1 = MAX_JNT
min_idx1 = 0
for i in range(0, N):
    # S is same as S. C: replicates C N times:
    # C[i, j] = S[i, j]
    C[i, j] = S[i, j]
    # computes all "at once"
    d[i, j] = SUB(SIMD(S[i, i + D] + j, (i, N, j, D)) # C[i, j] = C[j]
    p[i, j] = MUL(SIMD(d[i, j], d[i, j]))
    for j in range(0, D):
        # sum2[i], sum3[i], sum4[i] are size—N vectors
        sum2[i] = PHI(sum2[i], sum4[i])
        sum3[i] = ADD(sum2[i], sum4[i])
        t[i, j] = CMP(sum3[i], min_sum2[i])
        min_sum4[i] = MUX(t[i, j], sum3[i], min_sum2[i])
        min_idx4[i] = MUX(t[i, j], min_idx3[i], min_idx2)
    for i in range(0, N):
        min_sum2[i] = PHI(min_sum1[i], min_sum1[i])
        t[i] = CMP(sum3[i], min_sum2[i])
        min_sum4[i] = MUX(t[i], sum3[i], min_sum2[i])
        min_idx4[i] = MUX(t[i], min_idx3[i], min_idx2)
    return (min_sum2, min_idx2)

min_sum1 = MAX_JNT
min_idx1 = 0
for i in range(0, N):
    # S is same as S. C: replicates C N times:
    # S[i, j] = S[i, j]
    C[i, j] = S[i, j]
    # computes all "at once"
    d[i, j] = SUB(SIMD(S[i, i + D] + j, (i, N, j, D)) # C[i, j] = C[j]
    p[i, j] = MUL(SIMD(d[i, j], d[i, j]))
    for j in range(0, D):
        # sum2[i], sum3[i], sum4[i] are size—N vectors
        sum2[i] = PHI(sum2[i], sum4[i])
        sum3[i] = ADD(sum2[i], sum4[i])
        t[i, j] = CMP(sum3[i], min_sum2[i])
        min_sum4[i] = MUX(t[i, j], sum3[i], min_sum2[i])
        min_idx4[i] = MUX(t[i, j], min_idx3[i], min_idx2)
    for i in range(0, N):
        min_sum2[i] = PHI(min_sum1[i], min_sum1[i])
        t[i] = CMP(sum3[i], min_sum2[i])
        min_sum4[i] = MUX(t[i], sum3[i], min_sum2[i])
        min_idx4[i] = MUX(t[i], min_idx3[i], min_idx2)
    return (min_sum2, min_idx2)

Our results emphasize the opportunities 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 reviews the related work. §4 describes the cost estimation model and argues intractability of optimal scheduling. §5 details the frontend of the compiler, §6 focuses on MPC-IR and backend-independent vectorization, and §7 briefly describes translation into MOTION and MP-SPDZ. §8 presents the experiments and §9 concludes the paper.

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.

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 $\beta$ for a local MPC operation (e.g., XOR in Boolean sharing or ADD in Arithmetic sharing) and cost $\alpha$ for a remote MPC operation (e.g., MUX, CMP, etc.). Assuming that ADD is $\beta$ and SUB, CMP and MUX are $\alpha$, the MPC-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 $N \ast D$ 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: $\alpha = \alpha_{\text{fix}} + \alpha_{\text{var}}$. This gives rise to the following formula for amortized cost: $f(n) = n\alpha_{\text{fix}} + n\alpha_{\text{var}}$, as opposed to unamortized cost $g(n) = n\alpha_{\text{fix}} + n\alpha_{\text{var}}$. (We extend the same reasoning to $\beta$-instructions.)

Thus, the fixed cost of the vectorized program amounts to $2\alpha_{\text{fix}} + D\beta_{\text{fix}} + N(3\alpha_{\text{fix}})$. 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, $2\alpha_{\text{fix}} + D\beta_{\text{fix}} + N(3\alpha_{\text{fix}}) \ll ND(2\alpha_{\text{fix}} + \beta_{\text{fix}}) + N(3\alpha_{\text{fix}})$. 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 Related Work

3.1 MPC Compilers.

The early MPC compilers—Fairplay [BNP08] and Sharemind [BLW08]—demonstrated that MPC can be brought to applications. This led to the development of a diverse landscape of implementations/compilers for specific MPC protocols, e.g., PICCO [ZSB13], Obliv-C [ZE15], TinyGarble [Son+15], Wystria [RHH14a], Frigate [Moo+16], SPDZ/SPDZ-2 [Dam+12; Dam+13; KSS13], SCALE-MAMBA [COS19] and others. A new generation of state-of-the-art (and actively developed) MPC backends includes MP-SPDZ [Kel20] and the ABY/HyCC/MOTION [DSZ15; B¨ us+18; Bra+22] frameworks. These frameworks are highly parameterizable and allow the use of different MPC protocols/paradigms. Another recent development is Viaduct, a language and compiler that supports a range of secure computation frameworks, including MPC and ZKP. For a (by now slightly outdated) review of MPC compiler frameworks please see [Has+19]. In contrast to the above works, we focus on backend-independent optimizations available at the “higher-level” MPC-IR. Our key goal is to demonstrate provably correct compiler optimization for MPC, not a language frontend. Indeed, replacing our frontend with a more functional frontend, such as Viaduct’s, is an interesting future direction.

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. Similarly, GraphCG [Nay+15] introduces high-level programming abstractions and novel parallel oblivious algorithms; programmers can make use of the abstractions to write highly parallel and efficient secure algorithms.

HyCC [Biis+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 et al. [BK15] and [Büs18], uses an off-the-shelf source-to-source polyhedral compiler \(^1\) to perform vectorization at the level of source code. The disadvantage of using an off-the-shelf source-to-source compiler is that it solves a more general problem than what MPC presents and may forgo optimization because polyhedral compilation does not work well with conditionals (see [Ben+10]). In addition, the MPC-IR produced by our vectorization algorithm can serve as input to protocol mixing algorithms, such as, for example, OPA [IMZ19], which requires vectorized input and appears to use ad-hoc and manual vectorization.

### 3.2 Intermediate representations for MPC

Recent work makes progress on intermediate representations for MPC. Ozdemir et al. [OBW20] develop CirC, an IR with backends into ZKP and SMT primitives. MPC-IR is a higher level of representation than CirC; specifically, it does not unroll loops, leading to more scalable analysis. Heldmann et al. [Hel+21] present an LLVM-IR based toolchain for compilation into circuits. We take a different approach — rather than reverse engineer LLVM-IR, which is rich and complex, we propagate necessary information from the frontend to the middle-end IR; in addition, we formalize MPC-IR to enable reasoning about correctness of transformations. Another recent IR, developed concurrently and independently from ours, is FUSE [Bra+23]. It enables optimizations such as vectorization, in a way similar to the Section 6.1 optimization in MP-SPDZ [Kel20] (to our understanding). MPC-IR is a higher level IR and is orthogonal to FUSE. One can compile MPC-IR into FUSE and take advantage of the optimizations available at this level; our results with MP-SPDZ demonstrate that the two IRs and optimizations complement each other. In addition, we present syntax and semantics of MPC-IR, which enables reasoning about code transformations, a feature not present in [Bra+23; Hel+21].

### 3.3 Automatic vectorization in HPC

Automatic vectorization is a longstanding problem in high-performance computing (HPC). We present a vectorization algorithm for MPC-IR that builds upon classical loop vectorization by Allen and Kennedy [AK87]. We view Karrenberg’s work on Whole function vectorization [Kar15] as most closely related to our work — 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. The models can drive optimizations such as vectorization and protocol mixing; the optimizations interact 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, and the corresponding dependence analysis. Karrenberg’s work [Kar15] is rare in that space, in the sense that it considers vectorization over SSA, which has similarities with analysis over MPC-IR. We differ in the array representation, notion of dependence, and reasoning about dependence, as we specifically target MPC-IR.

### 4 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.

---

\(^1\)To our understanding, HyCC uses Par4All (https://github.com/Par4All/par4all), however, does not appear to be included with HyCC’s publicly available distribution.
4.1 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 \( \beta \) and a remote instruction (i.e. MUL) has cost \( \alpha \), where \( \alpha >> \beta \). We assume that all remote instructions have the same cost \( \alpha \) and all local ones have the same cost \( \beta \).

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 Andahl’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, \( n\alpha = \frac{p}{s}p\alpha + n(1 - p)\alpha \). For the purpose of the model we assume that \( s \) is large enough and the term \( \frac{p}{s}p\alpha \) amounts to a fixed cost incurred regardless of whether \( n \) is 10,000 or just 1. (This models the cost of preparing and sending a packet from party A to party B for example.) Therefore, amortized execution of \( n \) operations is \( f(n) = \alpha_{fix} + n\alpha_{var} \) in contrast to unamortized execution \( g(n) = n\alpha_{fix} + n\alpha_{var} \). We have \( \alpha_{fix} << n\alpha_{fix} \) and since fixed cost dominates variable cost (particularly for remote operations), we have \( f(n) << g(n) \).

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 \( 2\alpha_{fix} + 2\alpha_{var} \), however a MUL and a MUX instruction still cost \( 2\alpha_{fix} + 2\alpha_{var} \) even when scheduled “in parallel”.

4.2 (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 \( S_i \) is an instruction, 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

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

(1)

where \( L_\alpha \) is the number of \( \alpha \)-instructions and \( L_\beta \) is the number of \( \beta \) ones. (We used this formula to compute the cost of the unrolled MPC Source program in \$2\).

The cost of schedule \( P \) is more interesting:

\[
\text{cost}(P) = \sum_{i=1}^{m} \text{cost}(P_i)
\]

(2)

Each \( P_i \) may contain multiple instructions, and \( \text{cost}(P_i) \) is amortized. Thus, according to our model \( \text{cost}(P_i) = \alpha_{fix} + |P_i| \alpha_{var} \) if \( P_i \) stores \( |P_i| \) \( \alpha \)-instructions, or \( \text{cost}(P_i) = \beta_{fix} + |P_i| \beta_{var} \) if it stores \( \beta \)-instructions. (Similarly, we used this formula to compute the cost of the Optimized MPC Source program in \$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

\[\footnote{This is not strictly true, but assuming it, e.g. as in [IMZ19; DSZ15; MR18], helps simplify the problem.}\]
Common Supersequence for the independent schedules. We formalize the argument below and show that the scheduling problem is NP-hard via a reduction to the Shortest Common Supersequence problem [Vaz10].

Proof. To prove that optimal scheduling is an NP-Hard problem, we consider the following convenient representation. An MPC program is represented as a set of sequences \( \{s_1, \ldots, s_n\} \) of operations. In each sequence \( s_i \) operations depend on previous operations via a def-use i.e. \( s_i[j], j > 1 \) depends on \( s_i[j - 1] \).

As an example, consider the MPC program consisting of the following three sequences, all made up of two distinct \( \alpha \)-instructions \( M_1 \) and \( M_2 \), e.g., \( M_1 \) is MUL and \( M_2 \) is MUX. The right arrow indicates a def-use dependence, meaning that the source node must execute before the target node:

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

The problem is to find a schedule \( P \) with minimal cost. For example, one such schedule for the sequences above is

\[
M_1(1)||M_2(2) ; M_1(2) ; M_2(1)||M_2(3) ; M_1(1)||M_1(2)||M_1(3) ; M_2(3)
\]

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 \( 5\alpha_{\text{fix}} + 9\alpha_{\text{var}} \).

The problem of finding a schedule \( P \) with a minimal \( \text{cost}(P) \) 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 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 \( s_1 \ldots s_n \). It is also a shortest common supersequence. The cost of \( \text{cost}(P) = L\alpha_{\text{fix}} + N\alpha_{\text{var}} \) 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 \( L' \). \( \text{cost}(P') < \text{cost}(P) \) since \( L'\alpha_{\text{var}} + N\alpha_{\text{var}} < L\alpha_{\text{var}} + N\alpha_{\text{var}} \), contradicting the assumption that \( P \) has the lowest cost. \( \square \)

4.2.1 Remark on the (Necessity of) Assumptions

Only Two Types of Instructions (\( \alpha \) and \( \beta \)) We use this assumption in NP-hardness proof only, and it comes w.l.o.g. as to prove NP-hardness for an arbitrary circuit (that uses higher level gates), one can first reduce this new circuit to another one that only uses MUL and ADD gates and then use our proof. Since the reduction is polynomial time, this proof will imply NP-hardness of optimal scheduling for the new (lower level) circuit. We stress that this assumption is w.l.o.g., for optimizations that actually use the cost tables, e.g., mixing, one can use a refined model and it will not affect the NP-hardness proof.

Only the Same Type of Instructions Benefit from Vectorization This is not strictly true, but assuming it, e.g. as in [IMZ19; DSZ15; MR18], helps simplify the problem. One might be able to get additional improvements by identifying parallelizable heterogenous instructions, but it is unclear how to do this in an automated compiler without unrolling loops (which goes a level of abstraction below MPC-IR and would substantially slow down the compilation process and increase the memory needed by the compiler to store the unrolled loop information).

9
5 Compiler Frontend

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

5.1 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 \( \phi \)-nodes and MUX primitives (cf. §A.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\(^3\) 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. §6.4 and §A.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 \( \phi \)-node and the conditional that triggered the \( \phi \)-node. Specifically, given \( x = \phi(y, z) \), there is no information what conditional triggered the \( \phi \)-node and whether \( y \) corresponds to the true or false branch of the conditional. Moreover, \( \phi \)-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 \( \phi \)-nodes translate to MUX nodes, and loop \( \phi \)-nodes translate into what we call pseudo PHI-nodes (shown in lines 4 and 12 in Listing 1(b)).

\(^3\)A naive SSA translation will place a phi-node for every variable at every control merge node, but Cytron’s SSA places phi-nodes only where they are needed, at the Dominate Frontier of each variable assignment.
5.2 Syntax and Semantic Restrictions

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

\[
\begin{align*}
\text{expression} & ::= \text{e} \mid \text{op e} \mid x \mid \text{const} \mid A[e] \\
\text{sequence} & ::= s \mid s; s \\
\text{assignment stmt} & ::= x = e \mid A[e] = e \\
\text{for stmt} & ::= \text{for } i \text{ in range}(I) : s \\
\text{if stmt} & ::= \text{if } e : s \text{ else: } s
\end{align*}
\]

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 \(0 \leq i < I\) 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. However, it restricts write access to canonical writes, i.e., \(A[i, j, k] = \ldots\) where \(i, j, k\) 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

\[
\begin{align*}
1 \text{ if } e : A[i] = \text{val} \\
\text{Instead they write} \quad 1 \text{ if not(e): val = A[i]} \\
2 \text{ A[i] = val}
\end{align*}
\]

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 §A.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 \(A[i, j, k]\) 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.

6 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]).

6.1 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 $\phi$-nodes with the source $X$ in a loop nested into $Y$’s loop (not necessarily immediately).

- mixed forward edge $X \to Y$. $X$ is in some loop $i, j, k$ and $Y$ is in a loop nested into $i, j, k’$. We transform mixed forward edges as follows. Let $x$ be the variable defined at $X$. We add a variable and assignment $x' = x$ immediately after the $i, j, k$ loop. Then we replace the use of $x$ at $Y$ with $x'$. This transforms a mixed forward edge into an "inner-to-outer" forward edge followed by an outer-to-inner forward edge. Thus, Basic Vectorization handles one of "same-level", "inner-to-outer", or "outer-to-inner" def-use edges.

Closures We define $\text{closure}(n)$ 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 $\text{closure}(n)$
- $X$ is in $\text{closure}(n)$ if there is a same-level path from $n$ to $X$, and $X \to n$ is a same-level back-edge.
- $Y$ is in $\text{closure}(n)$ if there is a same-level path from $n$ to $Y$ and there is a same-level path from $Y$ to some $X$ in $\text{closure}(n)$.

6.2 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 $N \times D$ 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 $\text{raise\_dim}$ function expands a scalar (or array). There are two versions of $\text{raise\_dim}$. 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. 5.2):

$$\text{raise\_dim}(A, f(i, j, k), ((i:I), (j:J), (k:K)))$$
It produces a new 3-dimensional array $A'$ by iterating over $i, j, k$ and setting each element of $A'$ as follows:

$$A'[i, j, k] = A[f(i, j, k)]$$

The end result is that uses of $A[f(i, j, k)]$ in loop nest $i, j, k$ are replaced with canonical read-accesses to $A'[i, j, k]$ that can be vectorized. In the running Biometric example, $C' = raise\_dim(C, j, (i : N, j : D))$ lifts the 1-dimensional array $C$ into a 2-dimensional array. The $i, j$ loop now accesses $C'$ in the canonical way, $C'[i, j]$.

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 $i$:

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

**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 $j, k$:

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

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 $k$:

$$A'[i] = A[i, J - 1, K - 1]$$

**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 $raise\_dim$ when a scalar gives rise to an outer-to-inner dependence edge (e.g., $\text{sum!2}$ in line 6 of the MPC-IR code will be raised to a 1-dimensional array since $\text{sum!2}$ is used in the inner $j$-loop). We will $drop\_dim$ when a scalar gives rise to an inner-to-outer dependence edge (e.g., $\text{sum!2}$ 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 correctness reasoning.

### 6.3 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 §6.5. (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: \( x = Op(y_1, y_2) \) in loop \( i, j, k \) do

  for each argument \( y_n \) do

    case \( stmt' \) (def of \( y_n \)) \( \rightarrow stmt \) (def of \( x \)) of

      same-level: \( y_n' \) is \( y_n \)

      outer-to-inner: add \( y_n'[i, j, k] = \text{raise}_{\text{dim}}(y_n) \) at \( stmt' \) (more precisely, right after \( stmt' \))

      inner-to-outer: add \( y_n'[i, j, k] = \text{drop}_{\text{dim}}(y_n) \) at \( stmt \) (more precisely, in loop of \( stmt \) right after loop of \( stmt' \))

  end for

{ Optimistically vectorize all. \( I \) means vectorized dimension. } change to \( x[I, J, K] = Op(y_1'[I, J, K], y_2'[I, J, K]) \)

end for

{ Phase 2: Recreating for-loops for cycles; vectorizable stmts hoisted up. }

for each dimension \( d \) from highest to 0 do

  for each PHI-node \( n \) in loop \( i_1, ..., i_d \) do

    compute closure(\( n \))

  end for

{ \( cl_1 \) and \( cl_2 \) intersect if they have common statement or update same array; "intersect" definition can be expanded }

while there are closure \( cl_1 \) and \( cl_2 \) that intersect do

  merge \( cl_1 \) and \( cl_2 \)

end while

for each closure \( cl \) (after merge) do

  create for \( i_d \) in ... loop

  add PHI-nodes in \( cl \) to header block

  add target-less PHI-node for \( A \) if \( cl \) updates array \( A \)

  add statements in \( cl \) to loop in order of dependences

  { Dimension is not vectorizable: }

  change \( I_d \) to \( i_d \) in all statements in loop

  treat for-loop as monolith node for def-uses: e.g., some def-use edges become same-level.

end for

for each target-less PHI-node \( A_1 = \text{PHI}(A_0, A_k) \) do

  in vectorizable stmts, replace use of \( A_1 \) with \( A_0 \)

  discard PHI-node if not used in any \( cl \), replacing \( A_1 \) with \( A_0 \) or \( A_k \) 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 \texttt{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 \texttt{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 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.

6.4 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 §4. 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 §4. The iterative MPC-IR gives rise to what we called sequential schedule where loops are unrolled and MPC-IR with vectorized dimensions follows 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). $x[i, J, k]$ 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) $x[i, J, k]$ is enclosed in for-loops on non-vectorized dimensions i and k:

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

Linearization  Linearization is the concretization operation, which, as we mentioned earlier computes a schedule. The concretization function $\gamma$ 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 op_SIMD statement expands the vectorized dimension(s) into parallel statements; || introduces SIMD (parallel) execution. The concretization of the for i 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:

```
1 for j in range(0, D):
2 sum[3][j] = PHIL(sum[2][j], sum[4][j+1])
3 sum[4][j] = ADD(sum[3][j], p[j])
```

![Figure 3: MPC-IR Syntax and Semantics](image)

**Figure 3:** MPC-IR Syntax and Semantics. $\gamma$ defines the semantics of MPC-IR which is a linearization of input MPC-IR. A SIMD operation parallelizes operations across the vectorized J dimension. || denotes parallel execution, which is standard. $\gamma$ of a for loop unrolls the loop. ; denotes sequential execution. Iterative MPC-IR trivially extends to non-vectorized dimensions over the enclosing loops.
Assuming \( D = 2 \) and \( I = 2 \) for simplicity, linearization produces the following schedule:

```
1  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])
2  ;
3  sum!4[0,0] = ADD(sum!3[0,0], p[0,0]) ||
4  sum!4[1,0] = ADD(sum!3[1,0], p[1,0])
5  ;
6  sum!3[0,1] = PHI(sum!2[0,1], sum!4[0,0]) ||
7  sum!3[1,1] = PHI(sum!2[1,1], sum!4[1,0])
8  ;
9  sum!4[0,1] = ADD(sum!3[0,1], p[0,1]) ||
10 sum!4[1,1] = ADD(sum!3[1,1], p[1,1])
```

Note that by definition of the pseudo PHI function, \( \text{PHI}(\text{sum!}2[0,0], \text{sum!}4[0,-1]) \) evaluates to \( \text{sum!}2[0,0] \) 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 §6.1): 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 \( s_0 \) to denote statements in the concrete schedule as well as \( s_0[i,j,k] \) to denote fully instantiated values of \( i, j, \) and \( k \), such as for example \( s_0[0,1,0] \). 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 \( s_0 \rightarrow s_1 \) (indexing implicit), is defined in the standard way as well: \( s_0 \) writes a location, say \( x[i,j,k] \), and \( s_1 \) 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 \( s[i,j,k] \in \gamma(a_0) \) iff \( s[i,j,k] \in \gamma(a_1) \) and (2) \( s_0 \rightarrow s_1 \in \gamma(a_0) \) iff \( s_0 \rightarrow s_1 \in \gamma(a_1) \) (indexing implicit).

Let us first 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.

The main theorem below states that Basic vectorization preserves def-use edges.

**Theorem 1.** \( \gamma(a_0) \equiv \gamma(a_1) \).
Proof. Proof sketch of Theorem 1. 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 $\gamma(a_1)$; the key is that Basic vectorization preserves the def-uses in $a_0$.

A forward edge $s_0 \rightarrow s_1 \in a_0$ 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 $\gamma(a_0)$:

\[
\begin{align*}
& s_0[i] : s_1[i, j] & \text{outer-to-inner ordering} \\
& s_0[i, j] : s_1[i, j] & \text{same-level ordering} \\
& s_0[i, j] : s_1[i] & \text{inner-to-outer ordering}
\end{align*}
\]

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

Consider a backward edge $s_0 \rightarrow s_1 \in a_0$. 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, $s_0 \rightarrow s_1$ remains a backward edge in $a_1$. The linearization of the backward edge imposes ordering $s_0[i] : s_1[i]$ 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 may turn the backward edge into a forward one, however, it preserves the $s_0[i] : s_1[i]$ ordering constraint by construction.

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:

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

**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 $l[i, j, k]$ 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 $x[i, j, k] = y[i, j, k] \text{ or } z[i, j, k]$. By property $P$, there is the same statement in $\gamma(a_1)$ and it is scheduled after the definitions of $y[i, j, k]$ and $z[i, j, k]$. By the inductive hypothesis $y[i, j, k]$ and $z[i, j, k]$ hold the same values in $\gamma(a_0)$ as in $\gamma(a_1)$. Therefore, locations $x[i, j, k]$ hold the same value as well. We remark that due to the SSA form, each location $l[i, j, k]$ is defined at most once. For clarity, we elide PHI nodes and raising and dropping dimensions; extending def-use reasoning is straightforward.

### 6.5 Extension with Array Writes

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

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

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

```plaintext
for i in range(N):
A_1 = PHI(A_0, A_2);
B_1 = PHI(B_0, B_2);
C_2 = PHI(C_0, C_2);
D_1 = PHI(D_0, D_2);
A_2 = update(A_1, i, B_1[i] + 10);
B_2 = update(B_1, i, A_2[i] * D_1[i-1]);
C_2 = update(C_2, i, A_2[i] * D_2[i-1]);
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 $A_2 = \text{update}(A_1, i, B_1[i] + 10)$; 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 $\ldots = B_1[i]$ 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-$A$: $A_m[f(i, j, k)] = ...$ and use-of-$A$: $... = A_n[f'(i, j, k)]$ in loop $j$ do
        if $\exists i, j', k, k'$, s.t. $0 \leq i < I$, $0 \leq j, j' < J$, $0 \leq k, k' < K$, $i < j'$, and $f(i, j, k) = f'(i, j', k')$ then
            dep = True
        end if
    end for
    if dep == False then
        remove back edge from $A$’s φ-node in loop $j$.
    end if
end for

Consider a loop $j$ enclosed in some fixed $i$. Only if an update (definition) $A_m[f(i, j, k)] = ...$ at some iteration $j$ references the same array element as a use $... = A_n[f'(i, j, k)]$ at some later iteration $j'$, 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 $i, j, j', k, k'$ that satisfy $f(i, j, k) = f'(i, j', k')$. 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

\[
(0 \leq i < I) \land (0 \leq j, j' < J) \land (0 \leq k, k' < K) \land (i < j') \land f(i, j, k) = f'(i, j', k')
\]

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 $i$ and $i'$, where $i < i'$ that make $i = i'$ 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 $i < i'$ that make $i = i' - 1$ 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 $A[i, j] = ...$ 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 way as in the basic case, thus allowing us to extend correctness reasoning from...

18
the basic case. We will look to relax the restriction in future work, when additional optimizations are overlaid to vectorization (cf. §9).

7 Compiler Backends

Translation into the MOTION and MP-SPDZ backends is done by standard interpretation (i.e., syntax-directed translation) over the MPC-IR syntax shown in Fig. 3.

Naturally, translation into C++ code for the MOTION framework presents more challenges. First, 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 §C.1) according to the rules of §5.2. Another challenge is dealing with public values e.g., constants. Since there is no support 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 a performance hit. Instead, we keep shared copies of plain (public) variables, and update them in lock-step with updates to plain variable. When a plain variable is needed in a shared context, we use the shared copy.

8 Evaluation and Analysis

We evaluate on the two major MPC backend frameworks, MOTION and MP-SPDZ. Specifically, we compare the iterative (non-vectorized) version of a benchmark against the vectorized one. We compare on a benchmark suite that includes standard MPC benchmarks [Bü+18; IMZ19] as well as new ones from [FN21]. We present a list of the programs used for benchmarks in Appendix D.1. Note that, since we work at the higher level of MPC-IR, the source-to-source compilation overhead of our compiler is tiny. Even the slowest benchmark takes \( \approx 65 \text{ms} \) (average of 100 runs) to compile to either MP-SPDZ or MOTION. Therefore, we do not mention our compilation time separately.

8.1 MOTION Experiments

8.1.1 Experiment Setup

Experiment hardware is generously provided by CloudLab [Dup+19]. For the network, we consider two settings, LAN and 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. For WAN, we use a c6525-25g machine (located in Utah, US) and a c220g1 machine (located in Wisconsin, US). 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.

We run 2PC and 3PC experiments on LAN with input datasets that range in size from 2 to 4096. In the WAN setting we only perform 2PC experiments to save time; as evidenced by LAN experiments in §8.1.2, 3PC experiments would only take longer to run. 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. Due to space constraints, we present a subset of all results; additional experiments can be found in Appendix D. Tables and graphs in this section are with the largest dataset for which the non-vectorized run completes (it typically runs out of memory while the vectorized runs continue).

8.1.2 Results and Analysis

A summary of the effects of vectorization (MOTION backend) is presented in Table 2. In addition we show graphically circuit evaluation times in Fig. 4 and communication size in Fig. 5a. In terms of amenability to vectorization, we divide benchmarks into 3 categories: (1) High: these include Convex Hull, Cryptonets
Table 2: Non-vectorized vs. 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 name. All metrics are produced by MOTION.

| Benchmark             | GMW |          |          |          |          |          |          |          |          |          |          | BMR        |          |          |          |          |          |          |          |          |          |
|-----------------------|-----|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| Biometric Matching    | 146 | 16       | 1,784    | 119      | 1,413    | 140       | 89       | 263      | 1,595    | 139      | 2,716    | 312       | 12       | 4        | 34       | 28       | 14       |
| 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       | 20       | 167      | 16       | 304      | 59        |
| Cryptonets (Max Pooling) | 50  | 11       | 688      | 46       | 554      | 55        | 36       | 89       | 668      | 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    | 265       |
| 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      | 53       | 665      | 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       |
(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. (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). Fewer 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, Longest 102 and Max distance suffer a slowdown of 0.5x. There is opportunity for vectorization in these benchmarks according to our analytical model, particularly, there is a large EQ 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 the depth; also, EQ is relatively inexpensive in Boolean GMW and BMR compared to ADD and MUL, which also de-emphasizes the benefit of vectorization. We propose a simple heuristic (although we do leave all the benchmarks in the table): if the transformation increases circuit depth beyond some threshold (e.g. more than 10% of the original circuit), we can reject the transformation. In some settings it may still be desirable to vectorize e.g. in data constrained environments, as vectorization reduces communication costs.

Figure 4: 2PC: Circuit Eval. Time (Setup + Online) in Seconds, LAN Setting. Error-bars are Std. Deviation.

As shown in Fig. 5a, vectorization reduces communication, up to 12x in GMW and 3x in BMR (see §8.1.3 for details). In summary, vectorization enables better packing and it impacts an interactive protocol like GMW more than a constant round protocol like BMR. Fig. 5c 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. 5b) for vectorized circuits is a small fraction of the generation time for non-vectorized circuits (up to 200x less in GMW, 80x less in BMR). Online time and setup time are presented in Fig. 5d, and Fig. 5e.

---

4We remark that summation can be parallelized as a log(N)-depth tree (e.g., as in [Büs18]). This is an instance of the divide-and-conquer paradigm [FN21] and we have chosen to leave it as future work — divide-and-conquer can be applied more generally and in a principled way in conjunction with vectorization, rather than just on summation.
respectively.

Figures 6, 7a, and 7b zoom into Biometric matching. For input size beyond $N=256$ the memory usage exceeds available memory and prevents circuit generation for the non-vectorized case. Vectorization improves all metrics. In circuit evaluation (see Fig. 6), BMR sees higher speedup (23x faster) compared to GMW (10x faster), while GMW sees faster circuit generation time at 45x lower (see Fig. 7b) compared to BMR’s which is 35x lower. Communication size reduction (see Fig. 7a) is higher for GMW (10x less) compared to BMR (2.5x less). The number of gates, communication size and circuit generation time remain the same in the WAN setting. As expected, Setup and Online times increase, as shown in Fig. 8.

For the 3PC, we observe that, as expected, evaluation time (Fig. 9a) is higher than the 2PC. This is a direct consequence of higher online time (Fig. 9e) for the GMW protocol. Online time for BMR remains roughly the same, which is expected because the online phase is essentially local in BMR. BMR suffers a slowdown in the setup phase (Fig. 9f) however. This is due to the circuit for 3 parties requiring more computation. For brevity, we do not include graphs for (1) circuit generation time, (2) gates count, and (3) communication size for 3PC. Circuit generation sees a slowdown in BMR for the reason mentioned above,
communication size per channel and gates count remain the same. The experiments for 3PC confirm that adding more parties to an MPC increases resource requirements.

8.1.3 Analysis: Communication Size Reduction

As shown in communication size graph (Fig. 5a), 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 be 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 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 $\text{size}(h) + N \cdot \text{size}(p)$ 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\(^5\) in UDP compared to the at least 20 bytes\(^6\) in TCP. Both protocols are typically implemented over Internet Protocol and the header size of an IPv4 packet is 20 bytes\(^7\). In the Arithmetic GMW protocol, multiplication operation (MUL) is typically implemented using Beaver’s triples \([\text{Bea92}]\). This means that, in the online phase, all parties need to send $2\ell$ 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) 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.

### 8.2 MP-SPDZ Experiments

The MOTION compiler does not automatically apply vectorization; hence our MOTION experiments compare our vectorization optimization to “no optimization” (the iterative version of the benchmark) and, as expected, demonstrate high gains. Instead, MP-SPDZ does use vectorization but at a lower level than

---

ours [Kel20, Section 6.1] — the MP-SPDZ compiler unrolls the loop and merges (same) instructions from different iterations that may occur in parallel into a vector; the VM runs the vectorized instruction. Thus, it is reasonable to ask if our backend-independent optimization is compatible with MP-SPDZ and whether or not it still yields improvements. Our relevant experiments discussed below answer both these questions to the affirmative.

8.2.1 Experiment Setup

We run a representative subset of our MOTION experiments in the LAN setting (c.f. LAN configuration above). We remark that having already demonstrated the benefits of our optimization on a range of experiments with MOTION, the goal of our MP-SPDZ experiments is to address the above questions. Therefore, we chose a subset of benchmarks to run with MP-SPDZ that give a clear answer to the questions above while also demonstrating the delicate points of applying our optimizer on a back-end which already vectorizes (MP-SPDZ) vs. one that does not (MOTION). In particular, we focus on 2PC with semi-honest adversary for protocols following all three MPC paradigms (Boolean/Arithmetic GMW and BMR). As an

Figure 9: 3 Party LAN Setting: Various Matrices for Benchmark-Set, Error Bars are Standard Deviation.
Table 3: MP-SPDZ Compilation/Execution Stats, Sec. Param=40, Bit Length=32, 2PC 10Gbps LAN. Compilation and Run Time are in seconds, Peak Memory (Memory) and Communication (Comm.) are in MBs, Vectorized benchmarks have postfix (V) in their name.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Arithmetic</th>
<th>Semi-bin</th>
<th>Semi-BMR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MASCOT</td>
<td>SOHO</td>
<td></td>
</tr>
<tr>
<td>Biometric Matching</td>
<td>15.66</td>
<td>587.25</td>
<td>64.512</td>
</tr>
<tr>
<td>Biometric Matching (V)</td>
<td>16.16</td>
<td>605.53</td>
<td>64.512</td>
</tr>
<tr>
<td>Convex Hull</td>
<td>3.77</td>
<td>109.37</td>
<td>1,000.848</td>
</tr>
<tr>
<td>Convex Hull (V)</td>
<td>3.78</td>
<td>148.31</td>
<td>1,000.848</td>
</tr>
<tr>
<td>Count 10i</td>
<td>0.84</td>
<td>44.32</td>
<td>98.816</td>
</tr>
<tr>
<td>Count 10i (V)</td>
<td>0.94</td>
<td>51.88</td>
<td>98.816</td>
</tr>
<tr>
<td>Count 102</td>
<td>0.81</td>
<td>42.38</td>
<td>98.816</td>
</tr>
<tr>
<td>Count 102 (V)</td>
<td>0.94</td>
<td>51.96</td>
<td>98.816</td>
</tr>
<tr>
<td>Cryptonets (Max Pooling)</td>
<td>1.85</td>
<td>74.26</td>
<td>371.712</td>
</tr>
<tr>
<td>Cryptonets (Max Pooling) (V)</td>
<td>2.04</td>
<td>88.11</td>
<td>371.712</td>
</tr>
<tr>
<td>Database Join</td>
<td>8.62</td>
<td>244.82</td>
<td>786.432</td>
</tr>
<tr>
<td>Database Join (V)</td>
<td>9.56</td>
<td>281.16</td>
<td>786.421</td>
</tr>
<tr>
<td>Inner Product</td>
<td>0.14</td>
<td>24.18</td>
<td>512</td>
</tr>
<tr>
<td>Inner Product (V)</td>
<td>0.25</td>
<td>28.77</td>
<td>512</td>
</tr>
<tr>
<td>Longest 10i</td>
<td>16.32</td>
<td>594.25</td>
<td>161.792</td>
</tr>
<tr>
<td>Longest 102 (V)</td>
<td>10.12</td>
<td>601.04</td>
<td>161.792</td>
</tr>
<tr>
<td>Max Dist. Int. Symbols</td>
<td>15.40</td>
<td>573.94</td>
<td>94.720</td>
</tr>
<tr>
<td>Max Dist. Int. Symbols (V)</td>
<td>15.21</td>
<td>573.94</td>
<td>94.720</td>
</tr>
<tr>
<td>Minmax Points</td>
<td>3.35</td>
<td>103.69</td>
<td>991.360</td>
</tr>
<tr>
<td>Minmax Points (V)</td>
<td>3.77</td>
<td>158.95</td>
<td>991.360</td>
</tr>
<tr>
<td>MNNST ReLU</td>
<td>8.84</td>
<td>212.63</td>
<td>991.232</td>
</tr>
<tr>
<td>MNNST ReLU (V)</td>
<td>6.08</td>
<td>233.02</td>
<td>991.232</td>
</tr>
<tr>
<td>Private Set Intersection</td>
<td>11.01</td>
<td>288.23</td>
<td>1,048.704</td>
</tr>
<tr>
<td>Private Set Intersection (V)</td>
<td>9.36</td>
<td>325.30</td>
<td>1,048.704</td>
</tr>
</tbody>
</table>
extra point, to demonstrate that our optimization can also be applied to maliciously secure protocols, we run 2PC MP-SPDZ with the MASCOT protocol. Concretely, we run MP-SPDZ in the Arithmetic setting with the MASCOT and SOHO protocols, and in the Binary setting with the Semi-bin and Semi-BMR protocols. We run with datasets that trigger program complexity of at least $O(10^3)$. Our results are discussed below.

### 8.2.2 Results and Analysis

Table 3 summarizes our experiments. Vectorization has virtually no impact in the Arithmetic setting. In contrast, it has significant impact on compilation in the Binary setting, reducing compilation time from about 11% on Max Distance to 25x on Biometric. It has significant impact on memory footprint as well (22x in Biometric). We conjecture that the above discrepancy is due to the following reason: In the Arithmetic setting vectorization of data (typically integer arrays) through the API is as costly as merging of data and instructions by the MP-SPDZ compiler. Granularity of instructions is larger compared to the Binary setting, leading to fewer nodes and edges and lower complexity of the dependence graph and dependence analysis. In contrast, in the Binary setting’s finer granularity of instructions (e.g., multiplication of two integers is expressed in terms of multiple AND and XOR instructions) leading to a larger and denser graph and significantly more complex dependence analysis. Performing dependence analysis apriori on the very short MPC-IR, which is what our vectorization analysis does, informs the MP-SPDZ compiler and reduces analysis time and memory footprint.

We note in passing, that our experiments demonstrate that vectorization slightly increases the number of rounds on the regular expression programs (e.g., Count 10s). This is consistent with the MOTION results where we observed little improvement or slow down. Indeed, our vectorization algorithm breaks the single loop into several loops separating smaller loops with a vectorized operation, typically equality; we conjecture that the vectorized operation serves as a barrier preventing the MP-SPDZ compiler from merging instructions it would otherwise merge in the non-vectorized single-loop program (resulting in improved round complexity). Nevertheless, our LAN experiments show that the above increase in rounds is not reflected as slowdown on running time, so the significant compilation savings make our optimization worthwhile.

### 9 Conclusion and Future Work

We presented a formalization of the MPC-IR intermediate language followed by a specific backend-independent optimization at the level of MPC-IR: novel SIMD-vectorization. We demonstrated that vectorization has significant impact on performance and showcased the backend-independent nature of our optimization by applying it to two mainstream and parameterizable MPC frameworks, namely MOTION and MP-SPDZ.

We are excited about the opportunities for future work — integration with protocol mixing, divide-and-conquer reasoning and parallelization, as well as inter-procedural context-sensitive analysis at the level of MPC-IR will improve MPC programmability and efficiency.

### References


A

Compiler Frontend

A.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 §5.2. 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 §5.2). 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., `A[i] = x`, change the statement to `A = Update(A, i, x)`. If the statement is not already three-address code, for each sub-expression in the right-hand side of the assignment, insert an assignment to a temporary variable.

SSA conversion: Convert the 3-address CFG to SSA with Cytron’s algorithm.
A.2 From SSA to MPC-IR

Once the compiler converts the code to SSA, it transforms $\phi$-nodes that correspond to if-statements into MUX nodes. From the 3-address CFG conversion step, $\phi$-nodes corresponding to if-statements will be in a basic block with the merge condition property. For example, if $X!3 = \phi(X!1, X!2)$ is in a block with merge condition $C$, the compiler transforms it into $X!3 = \text{MUX}(C, X!1, X!2)$. 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 $\phi$-node has exactly two multiplexer arguments, which dovetails into MUX. This is in contrast with Cytron’s algorithm which operates at the level of the CFG and allows for $\phi$-nodes with multiple arguments.) Each time the compiler visits a block, it adds the block’s statements to the MPC-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
    end if

32
return result
end if
return search(entry block of the control-flow graph)

Now, the remaining $\phi$-nodes in MPC-IR are the loop header nodes. These are the pseudo $\phi$-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 $x!2$.

A.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 6 we extended the base syntax to account for vectorization.) Most notably, there is no if-then-else statement but there are MUX expressions:

$$e ::= e \text{ op } e | x | \text{ const } | A[e] | \text{ MUX}(e,e,e)$$

$$s ::= s; s$$

$$x = \text{ PHI}(x,x) | x = e | A[e] = e$$

$$\text{ for } i \text{ in range}(I) : s$$

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

$$\tau ::= \text{ int } | \text{ bool } | \text{ list[int] } | \text{ list[bool]}$$

$$q ::= \text{ shared } | \text{ plain }$$

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 $\ll$ 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:

$$\Gamma \vdash e_1 : \langle q_1 \tau \rangle$$

$$\Gamma \vdash e_2 : \langle q_2 \tau \rangle \quad \tau \in \{\text{int}, \text{bool}\}$$

$$\Gamma \vdash e_1 \text{ op } e_2 : \langle q_1 \lor q_2 \tau \rangle$$

$$\Gamma \vdash e_1 : \langle \text{ plain } \text{ int} \rangle \quad \Gamma \vdash A : \langle q \text{ list[}\tau] \rangle$$

$$\Gamma \vdash A[e] : \langle q \tau \rangle$$

$$\Gamma \vdash e_1 : \langle \text{ int } \rangle \quad \Gamma \vdash e_2 : \langle q_2 \tau \rangle \quad \Gamma \vdash e_2 : \langle q_2 \tau \rangle$$

$$\Gamma \vdash \text{ MUX}(e_1,e_2,e_3) : \langle q_1 \lor q_2 \lor q_3 \tau \rangle$$

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.

$$\Gamma \vdash x_1 : \langle q_1 \tau \rangle \quad \Gamma \vdash x_2 : \langle q_2 \tau \rangle \quad \Gamma \vdash x_3 : \langle q_3 \tau \rangle$$

$$\Gamma \vdash x_1 = \text{ PHI}(x_2,x_3) : OK$$

$$\Gamma \vdash x \in \langle q_1 \tau \rangle$$

$$\Gamma \vdash e : \langle q_2 \tau \rangle$$

$$\tau \ll \tau \in \{\text{int}, \text{bool}\}$$

$$\Gamma \vdash x = e : OK$$
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 whether a statement operates on shared variables or plaintext ones, and the backend generates appropriate MOTION code.

B Backend-Independent Vectorization

This appendix contains the additional content corresponding to §6. The algorithm for basic vectorization is listed and described in §6.3, here we explain it further via its application to the running example of the paper, Biometric Distance.

B.1 Example: Biometric

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

```plaintext
1 min_sum1 = MAX_INT
2 min_idx1 = 0
3 for i in range(0, N):
4    min_sum2 = PHI(min_sum1, min_sum4)
5    min_idx2 = PHI(min_idx1, min_idx4)
6    sum2 = 0
7    for j in range(0, D):
8        sum3 = PHI(sum2, sum4)
9        d = SUB(S[(i * D) + j], C[j])
10       p = MUL(d,d)
11       sum4 = ADD(sum3, p)
12       t = CMP(sum3, min_sum2)
13       min_sum3 = sum3
14       min_idx3 = i
15       min_sum4 = MUX(t, min_sum3, min_sum2)
16       return (min_sum2, min_idx2)
```

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., \(\text{min} \sum 3 = \text{sum} 3\)) are implicitly vectorized. The example illustrates the two different versions of \(\text{raise} \_\text{dim}\). E.g., \(\text{raise} \_\text{dim}(C, j, (i:N, j:D))\) reshapes the read-only input array. \(\text{drop} \_\text{dim}(\text{sum} 3)\) removes the \(j\) dimension of \(\text{sum} 3\).
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 $\text{sum}!3 = \text{PHI}(...)$, the algorithm first computes its closure. The closure amounts to the phi-node itself and the addition node $\text{sum}!4 = \text{ADD}(\text{sum}!3,p!3)$, accounting for the loop-carried dependency of the computation of $\text{sum}$. 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 $p[I,J] = \text{MUL}(d[I,J],d[I,J])$ to the monolithic for-loop and from the for-loop to $\text{sum}!3 \leftarrow \text{dropdim}(\text{sum}!3)$. 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:

When processing the outer loop two closures arise, one for $\text{min\_sum}!2[I] = \text{PHI}(...) \text{ and one for } \text{min\_idx}!2[I] = \text{PHI}(...)$. Since the two closures do not intersect, we have two distinct for-loops on $i$:
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:

Phase 4 of Basic Vectorization

This phase adds SIMD operations:
B.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:

```python
for i in range(0, N):
    A_1 = PHI(A_0, A_2)
    B_1 = PHI(B_0, B_2)
    C_1 = PHI(C_0, C_2)
    D_1 = PHI(D_0, D_2)
    A_2 = update(A_1, i, B_1[i] + 10);
    B_2 = update(B_1, i, A_2[i] * D_1[i-1]);
    C_2 = update(C_1, i, A_2[i] * D_1[i-1]);
    D_2 = update(D_1, i, B_2[i] * C_2[i]);
```

Note that since all def-uses are same-level (i.e., reads and writes of the array elements) no raise dimension or drop dimension happens.

Phase 2 computes the closure of 5; cl = {5, 7, 8, 9} while 6 is vectorizable. Recall that 2, 3, and 4 are target-less phi-nodes. Since the closure cl includes updates to B and C, the corresponding phi-nodes are added back to the closure and the def-use edges are added back to the target-less nodes. The uses of A_1 and B_1 in the vectorized statement turn into uses of A_0 and B_0 respectively; this is done for all original target-less phi-node. (But note that A_0 is irrelevant; the update writes into array A_2 in parallel.) Finally, the target-less phi-node for A is discarded.

```python
for i in range(0, num_bins):
    res1 = PHI(res, res2)
    for j in range(0, N):
        res2 = PHI(res1, res3)
        tmp1 = [A[i] == i]
        tmp2 = [res2[i] + B[i]]
        tmp3 = [A[i] == j]
        tmp2 = [res3[i, j] + B[i]]
        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].
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)  # replacing res1 with res2^, 2nd arg. NOOP
    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))))

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

C Compiler Back End

MOTION framework requires that all variables are marked as plain or shared following the type system in §5.2. We require that only inputs are marked as either shared or plaintext, and infer qualifiers for other variables through taint analysis of §C.1. We provide details of code generation for MOTION backend in §C.2.

C.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 sum1!2 and sum1!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, ...]
sum1!1 = 0
for l in range(0, N):
    sum1!2 = PHI(sum1!1, sum1!3)
    sum1!3 = sum1!2 + plaintext_array[l]
1. \( A[i] = \text{val} \)

**IMP Source**

1. \( A1 = \text{update}(A1, i, \text{val}) \)

2. \( A_2 = A_1; \)

**MPC-IR**

Table 4: MOTION Translation: Array Updates

1. \( \text{MPC}_\text{PLAINTEXT}_i = 0; \)
2. \( \text{tmp} = \text{arr}[\text{MPC}_\text{PLAINTEXT}_i]; \)
3. \( \text{for } i \in \text{range}(N): \)
4. \( \text{if } (\text{MPC}_\text{PLAINTEXT}_i != 0) \{ \)
5. \( \text{tmp} = \text{val}_0; \)
6. \( \} \)
7. \( \ldots \)

**MOTION Code**

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.

### C.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.

**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 4.

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 5 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-dimensional representation of a SIMD value, along with the size of each dimension and the requested slice’s indices, and converts that slice to a MOTION SIMD value. Because MOTION supports SIMD operations using the same C++ operators as non-SIMD operations, we do not need to perform any other transformations to the expression. Therefore, once vectorized accesses are inserted the translation of an expression containing SIMD values is identical to that of expressions without SIMD values.

Similarly, the vectorized_assign() function assigns a (potentially SIMD) value to a slice of a vectorized array. This operation cannot be done with a simple subscript as SIMD assignments will update a range of values in the underlying array representation.

Updating SIMD arrays is also implemented differently from updating non-vectorized arrays. Instead of separating the array update from the assignment of the new array, these steps are combined with the vectorized_update() utility function. This function operates identically to vectorized_assign(), however it additionally returns the array after the assignment occurs. This value is then used for the assignment to the new variable. Listing 6 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 §6.2. 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: \( A'[n, m, d] = A[n, m] \) 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.

D Evaluation

D.1 Benchmarks List

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 (vectorized and non-vectorized experiments) and $N=4096$ for vec (i.e., only the vectorized experiment because the non-vectorized experiment runs out of memory for this input size). $D$ 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 $R \times C$-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 of sizes $A$ 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.


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 0s 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 \( A \), Bob holds a set of size \( B \) and they compute intersection. We use \( A=B=128 \) for both and \( A=B=1024 \) for vec.

### D.2 Comparison with MOTION-native Inner Product

![Graphs showing comparison](image)

Figure 10: 2 Party LAN Setting: Inner Product, x-axis lists vector size.

We compared our results on Inner Product (Fig. 10a, Fig. 10b and Fig. 10c) with the manually SIMD-ified one distributed with MOTION source. 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 multiplications are the same, however, our addition loop incurs significant cost (ADD is non-local and expensive in Boolean GMW). The MOTION-native loop runs

```c
result += mult_unsimplified[i];
```

while our compiler-generated loop runs

```c
result[i] = result[i-1] + mult_unsimplified[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.

It is unrealistic to expect that MOTION’s static analysis will detect the associative accumulation in the scalar expansion code above. This is because MOTION’s analysis is at source or AST level and such static analysis at the AST level is difficult. Our investigation showed that not only MOTION does not optimize

```c
result[i] = result[i-1] + mult_unsimplified[i];
```

it does not optimize the simpler accumulation:

```c
result = result + mult_unsimplified[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.