RESEARCH-ARTICLE

Coyote: A Compiler for Vectorizing Encrypted Arithmetic Circuits

RAGHAV MALIK, Purdue University, West Lafayette, IN, United States

KABIR SHETH, Purdue University, West Lafayette, IN, United States

MILIND KULKARNI, Purdue University, West Lafayette, IN, United States

Published: 25 March 2023

Citation in BibTeX format

ASPLOS '23: 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3
March 25 - 29, 2023
BC, Vancouver, Canada

Conference Sponsors:
SIGARCH
SIGOPS
SIGPLAN

Coyote: A Compiler for Vectorizing Encrypted Arithmetic Circuits

Raghav Malik
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
malik22@purdue.edu

Kabir Sheth
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
kdsheth@purdue.edu

Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
milind@purdue.edu

ABSTRACT

Fully Homomorphic Encryption (FHE) is a scheme that allows a computational circuit to operate on encrypted data and produce a result that, when decrypted, yields the result of the unencrypted computation. While FHE enables privacy-preserving computation, it is extremely slow. However, the mathematical formulation of FHE supports a SIMD-like execution style, so recent work has turned to vectorization to recover some of the missing performance. Unfortunately, these approaches do not work well for arbitrary computations: they do not account for the high cost of *rotating* vector operands to allow data to be used in multiple operations. Hence, the cost of rotation can outweigh the benefits of vectorization.

This paper presents Coyote, a new approach to vectorizing encrypted circuits that specifically aims to optimize the use of rotations. It tackles the scheduling and data layout problems simultaneously, operating at the level of subcircuits that can be vectorized without incurring excessive data movement overhead. By jointly searching for good vectorization and lane placement, Coyote finds schedules that avoid sacrificing one for the other. This paper shows that Coyote is effective at vectorizing computational kernels while minimizing rotations, thus finding efficient vector schedules and smart rotation schemes to achieve substantial speedups.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; *Software and application security*; • **Software and its engineering** → **Compilers**; **Domain specific languages**.

KEYWORDS

Homomorphic Encryption, Arithmetic Circuits, Vectorization

ACM Reference Format:
Raghav Malik, Kabir Sheth, and Milind Kulkarni. 2023. Coyote: A Compiler for Vectorizing Encrypted Arithmetic Circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582057

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) refers to any encryption scheme that allows for homomorphically adding and multiplying ciphertexts, so that the sum of the encryptions of two integers is an encryption of their sum, and similarly the product of the encryptions of two integers is an encryption of their product [11]. While FHE is a powerful technique for carrying out privacy-preserving computations on encrypted data, it has a major downside: it is slow. Homomorphic computations over ciphertexts are often orders of magnitude slower than a corresponding plaintext computation. Many FHE cryptosystems support packing large numbers of ciphertexts into *ciphertext vectors*, essentially compensating for the inherent slowness of FHE by enabling SIMD-style computation [2, 20]. To properly take advantage of ciphertext packing, we need a compiler that can vectorize arbitrary FHE programs.

Vectorizing compilers for FHE, such as CHET [8] and Porcupine [5], exist. However, neither of these approaches meets the need for a vectorizing compiler for arbitrary FHE programs. While CHET is optimized for highly regular computations over packed tensors (such as neural networks), it does not generalize to more irregular programs. Porcupine, which uses a synthesis-based approach to generate vectorized code for arbitrary kernels, does work for a more general class of programs. However, it is not automated, as it requires a programmer-provided *sketch* as a starting point.

Other approaches to vectorizing arbitrary, non-loop-based code, such as Superword-Level Parallelism [14], also fail here. SLP aggressively packs isomorphic instructions into vectors, because it assumes that shuffling vector lanes around or indexing into a vector is relatively cheap. In FHE, however, the vectors are not physical vector registers with slots for data: the only way to move data between vector lanes in FHE is by performing a cyclic rotation of the entire vector. Realizing the shuffles incurred by SLP with a series of masks and rotates is expensive, and can quickly outweigh any benefits from vectorizing.

More recent takes on SLP, such as VeGen [4] and goSLP [16], recognize the need to take the cost of data movement into account. VeGen can decide to not pack certain instructions together because the data movement cost incurred is not worth it. However, VeGen does its reasoning *locally*; that is, it cannot reason about the effect packing instructions together may have on shuffling costs much later in the program. This tradeoff, fine in circumstances when shuffling is relatively cheap, is inappropriate for FHE, where shuffling is very expensive. While goSLP does reason globally, the cost model it uses to avoid over-packing is incompatible with the semantics of FHE vectorization. We discuss this further in Section 7.

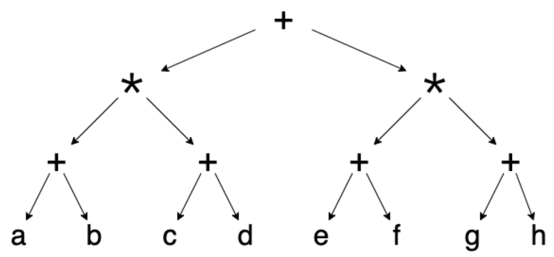


Figure 1: An example of an arithmetic circuit

```
%0 = a + b
%1 = c + d
%2 = %0 * %1
%3 = e + f
%4 = g + h
%5 = %3 * %4
%6 = %2 + %5
```

(a) No vectorization

```
[%0, %1, %3, %4] = [a, c, e, g] +
                   [b, d, f, h]
[%2, _, %5, _] = [%0, _, %3, _] *
                 [%1, _, %4, _]
[%6, _, _, _] = [%2, _, _, _] +
                [%5, _, _, _]
```

(b) Aggressive vectorization, incurs two rotates

```
[%1, %3] = [c, e] + [d, f]
[%0, %4] = [a, g] + [b, h]
[%2, %5] = [%0, %3] * [%1, %4]
[%6, _] = [%2, _] + [%5, _]
```

(c) Optimal schedule, incurs one rotate

Figure 2: Possible schedules for Figure 1

1.1 The Vectorization/Rotation Tradeoff

Without carefully accounting for, and controlling, rotation, traditional vectorization strategies can lead to *slowdowns*, rather than speedup, in an FHE setting. It is often beneficial to give up vectorization opportunities to avoid incurring expensive rotations.

Consider the arithmetic circuit in Figure 1 implementing $((a + b) * (c + d)) + ((e + f) * (g + h))$.[1] A naïve vectorization would pack together the four additions at the first level, and the two multiplies at the second level (as in the schedule in Figure 2b). The resulting schedule has a vector add, followed by a vector multiply, followed by an add. Rotations are needed between each operation to align the outputs of each operation with the next. Using an approximate model[2] of the relative latencies of each instruction in which multiplies and rotates have a latency of 1 and addition has a latency of 0.1, the total cost of this schedule is 3.2. However, by doing no vectorization and executing the circuit entirely with scalar operations (Figure 2a), we have five adds and two multiplies, with an overall cost of 2.5. In this case, vectorization actually makes the performance *worse*! Figure 2c shows how we can do better: We pack the $a+b$ and the $e+f$ adds separately from the $c+d$ and $g+h$ adds, so that neither of them require a rotation to align with the multiply above them. By saving one rotation at the cost of an extra vector addition, we get a schedule with an overall cost of only 2.3.

[1]We adopt the FHE-standard representation of arithmetic circuits as the intermediate representation for our programs [1, 5, 11, 15].

[2]Algorithms in HElib [12] assigns a "high latency" to both multiplies and rotates and a "low latency" to adds. For both simplicity and concreteness, we assume a 10:1 ratio between "high latency" and "low latency."

We need a new arbitrary vectorization strategy that is *FHE-aware*; i.e., it packs instructions without relying on regularity in the original computation, and can still account for the high cost of data movement throughout the program.

1.2 Co-optimization of Vector Packing and Data Layout

Our key insight is that because rotations are so expensive, data layout and vector packing are fundamentally intertwined. Rather than treat these as separate problems, we must optimize them *together* when finding a schedule. The main obstacle to optimality when using the classical approach is simple: vectorizing across instructions by aggressively packing them into vectors can require substantial and complex data movement to align operands for downstream vector instructions (e.g., SwizzleInventor [17], which resorts to sketch-based synthesis to generate the appropriate permutations). If permuting operands between lanes can only be done with expensive rotations, an aggressively packed schedule can incur so much overhead that no amount of vectorization makes it worth it.

Instead, we develop an approach that works at the level of *subcircuits*, splitting the program up into smaller pieces within which all the computations are locked into a single lane to avoid doing any rotations at all. While vectorizing across subcircuits gives up some packing potential (because operations within a subcircuit cannot be vectorized together), the savings on rotation costs can make up the difference: the subcircuits prevent over-vectorization that incurs too many rotations. The optimal schedule of Figure 2c can be viewed as grouping $(a + b)$ with its downstream multiply in one subciruit, and $(g + h)$ with its downstream multiply in another subcircuit, and then vectorizing those two subcircuits together.

This approach yields a natural question: how do we decide which computations to merge into a subcircuit? This seems circular: subcircuit merging is intended to yield fewer rotations, which are determined by data layout, and data layout is driven by which operations are vectorized together, which in turn is constrained by subcircuit identification.

1.3 Contributions

This paper presents Coyote, the first vectorizing, FHE-aware compiler for programs that do not have regular structure. Coyote breaks the circular dependence between vector packing and data layout by using an iterative process that alternates between making packing decisions and determining data layout. Coyote uses simulated annealing to find optimal data layouts, and uses these to guide a best-first search towards optimal vector packs. Crucially, Coyote uses layouts from previous iterations of scheduling to identify subcircuits that would be profitable to merge, and then re-schedules based on the new subcircuits.

Rather than incurring many expensive data shuffling operations by aggressively vectorizing the whole program, Coyote uses a subcircuit-based approach to scheduling and employs an FHE-specific cost model co-optimize the data layout and vector packing, producing a schedule that enjoys the benefits of vectorization while still being able to efficiently realize the necessary rotations. For example, Coyote's vectorization algorithm is able to correctly identify the schedule in Figure 2c as being optimal.

The specific contributions we make are:

(1) An algorithm for simultaneously searching the space of data layouts and the space of vector packings to find an efficient combination.
(2) A lightweight Python embedded DSL called Coyote, with a compiler that uses this algorithm to generate efficient FHE code for arbitrary programs

We tested Coyote by using it to compile six computational kernels (matrix multiply, point cloud distances, 1D convolution, dot product, sorting a list, and finding the maximum element of a list), and compared the performance of the vectorized code to to the original unvectorized code. We also randomly generated several irregular polynomial-evaluation programs to measure the effect of things like operation density on Coyote's ability to vectorize. We find that Coyote very effectively vectorizes programs, yielding efficient vector schedules with optimized rotations.

2 BACKGROUND

2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) refers to any encryption scheme with the property that encrypting inputs, computing over them, then decrypting the result is equivalent to computing over the un-encrypted inputs. [11] FHE is hence useful for computing on encrypted data, improving privacy in situations such as computation offloading. Since addition and multiplication are complete, FHE can be used to realize arbitrary functions on encrypted data.

The Brakerski/Fan-Vercauteren (BFV) [10] cryptosystem, which is the particular FHE scheme that we use in this paper, is based on the Ring Learning With Errors (RLWE) problem. Ciphertexts in BFV are represented as high degree polynomials with an "error term", which is a small amount of noise added to the polynomial to make the scheme "CPA-secure" (in other words, the same plaintext will not encrypt to the same ciphertext each time).

2.1.1 Limitations. While FHE is an attractive approach to performing privacy-preserving computation, it presents a few challenges. First, the polynomial encoding of ciphertexts incurs a huge overhead for any secure computation. To achieve a reasonable degree of security, the polynomials need to be quite large, so a single primitive ciphertext operation like an add or a multiply gets translated into very expensive polynomial math. This means that all but the smallest FHE applications are often too slow to be practically run.

A second challenge for FHE computation is related to the noise added to ciphertexts. When setting up an FHE computation, the encryption parameters are used to determine a safe *noise margin* for ciphertexts, which describes the level of noise above which ciphertexts can no longer be decrypted. Freshly encrypted ciphertexts are well below this margin, but multiplying two ciphertexts increases the amount of noise present in the result. BFV does support *bootstrapping*, which is a technique for homomorphically computing a fresh encryption of a ciphertext to "reset" its noise level; however, bootstrapping is an expensive procedure. When designing an FHE computation, therefore, it becomes important to limit its multiplicative depth to avoid bootstrapping as much as possible.

Finally, because of the nature of secure computation, FHE does not support branching over ciphertexts—conditionals cannot depend on the values of encrypted data, otherwise the path taken through the computation leaks information about the data. In particular, this precludes FHE computations from having any kind of control flow structures, including conditionals and loops, that are control-dependent on ciphertexts.

2.1.2 Arithmetic Circuits. Since FHE does not support loops or conditionals, computations have to be represented as combinatorial arithmetic circuits. In particular, these arithmetic circuits we work with closely resemble expression forests, where some of the trees may in fact be DAGs (directed acyclic graphs) if any inputs are used in multiple places. For the rest of this paper, we assume that the programs we are compiling are already expressed in this way, and talk about how to map the computations encoded as arithmetic circuits to vectors. In practice, this is not too restrictive, since any loop with known (plaintext) bounds can be fully unrolled, and any conditional branching on a ciphertext can be converted into a "mux" by evaluating both branches and only selecting the correct output. The frontend DSL of Coyote does exactly that by staging python programs, producing arithmetic circuits it can compile.

2.2 Vectorization

Single instruction, multiple data, or SIMD, is a way of amortizing the run-time complexity of a program by *vectorizing* it, or lifting its scalar computation to one that operates over packed vectors. To vectorize, we need to first find sets of isomorphic scalar instructions and then decide how to pack the scalar operands of those instructions into vectors before replacing all of them with a single vector instruction. In traditional SIMD, this process relies heavily on the presence of data-parallel loops in the original program. Unrolling the loop by a few iterations (usually four or eight) produces a set of isomorphic instructions, one for each unrolled iteration. These are then packed into vectors, with one iteration per vector slot, and lifted into vector instructions. Thus, a loop that performs a scalar computation N times can be lifted into one that performs a semantically equivalent vector computation $N/4$ times.

Superword-Level Parallelism (SLP) is a more general technique that does not rely on the presence of loop-based control structures in the program to find vectorizable instructions. SLP analyzes a whole sequence of scalar instructions at once, looking for sets of *available instructions* (instructions whose operands have already been scheduled) that are all isomorphic to each other. At each step, it picks such a set and packs its instructions together into a vector, scheduling them together.

Vectorization in FHE. The way BFV encodes ciphertexts allows abstractly representing large vectors of values as being encrypted into a single ciphertext. In particular, homomorphic operations on such ciphertexts correspond to element-wise operations on the underlying packed vectors [2]. These polynomial rings also have specific *automorphisms* that cyclically permute the "slots" into which elements are packed (hereafter called vector lanes). In other words, ciphertext packing allows us to turn FHE into an abstract SIMD architecture with instructions for (ciphertext) vector addition and multiplication, as well as vector rotation. This style of vectorization has a few peculiarities that distinguish it from normal vectorization:

(1) The vectors are much larger than traditional hardware vector registers (e.g. several thousand slots wide, compared to the usual 4 or 8 slots). Utilizing this much space poses unique challenges.
(2) Unlike with physical vector registers, there is no *indexing* primitive that can directly access a value in a particular slot of a ciphertext vector.
(3) In general, the only way to move data between vector slots is by rotating the entire vector. This makes it much more important to assign vector lanes to packed instructions optimally, since realizing arbitrary permutations by composing several rotations quickly gets computationally expensive.

The challenges posed by points (2) and (3) in particular preclude us from simply using SLP-style vectorization, since its local reasoning means it does not sufficiently consider the high cost of data movement between lanes when deciding what instructions to pack together. We discuss the specific drawbacks of applying the SLP approach to FHE in Section 7.

3 COYOTE OVERVIEW

Coyote provides an embedded DSL (eDSL) that allows programmers to use a high level language to express computations in FHE. This computation is translated into an arithmetic circuit representing the computation, which is then compiled into vectorized FHE code. The process of compiling a circuit into vectorized code is as follows:

3.1 Compilation Steps

This section gives an overview of how Coyote vectorizes an arbitrary arithmetic circuit using the process laid out in Figure 3. We will use the circuit in Figure 1 as a running example. Compilation proceeds as follows:

(1) Coyote *quotients* an input circuit (collapses subcircuits into single vertices) and assigns lanes to resulting vertices to produce a *pre-schedule* that can be realized into a more efficient vector program. The result is a graph whose vertices correspond to connected subgraphs of the original circuit, such that no two vertices at the same height have the same lane (and hence are eligible to be vectorized together). Coyote collapses a subcircuit when it determines that the overhead of internally vectorizing it is not worth the gain from vectorization, so this step essentially forces certain operations to happen in scalar on a single lane. Section 4.2 describes how Coyote makes this decision.
    In the example in Figure 4a, the circled pairs of vertices are collapsed, yielding the quotient circuit in Figure 4b. The lane assignment for this pre-schedule puts each un-quotiented addition on the same lane as its quotiented parent, and chooses one of these lanes on which to place the root of the tree.
(2) The (collapsed) vertices at each height are aligned to pack together isomorphic nodes, producing a vector schedule from the pre-schedule. In the example, the two adds at height 1 get trivially aligned, and the two "supernodes" at height 2 get aligned by packing together the two adds and the two multiplies. No alignment is needed for the single vertex at height 3. The details of the alignment procedure are given in Section 4.4. Figure 4c shows the result of this alignment.
(3) Coyote compiles the schedule into a vector IR. The crux of this compilation step is figuring out when to *blend* and *rotate*. When a vector operand requires values from several different instructions, Coyote emits code to "blend" the results together into a single vector. When the lane an operand is used in is different from the lane it was produced in, Coyote emits a rotation instruction to move the operand into the correct lane. Notice that each arc in the pre-schedule connecting vertices of different lanes corresponds to a rotation in the generated vector IR. Figure 4d shows the vector code Coyote generates for our running example. Notice that the generated code contains two blends and one rotate. The blends are necessary[3] because on line 3 of the schedule, %0 and %3 are used in the same vector despite being produced in two separate vectors. Since none of the operands need to shift lanes, the vector instruction t0 = blend(v0@10, v1@01) takes [%0, %4] and [%1, %3] and blends them together to produce [%0, %3], which is exactly the operand used on line 3. Coyote emits a rotation because %5 gets used on a different lane than it is produced. The vector instruction s0 = v2 ≫ 1 takes [%2, %5] in v2 and produces [%5, %2] in s0. Section 5.2 describes the specifics of code generation.

3.2 Using Coyote

A programmer can use Coyote's DSL (shown in Figure 5) to specify a computation and generate an arithmetic circuit. The DSL exposes a number of ways to annotate inputs to the computation:*replication, packing*, and fixing a*layout*. Annotating an input with "replicate" indicates that a copy of the input should be passed to the circuit for each place it is used (ensuring that each copy gets used exactly once). By default, inputs are unreplicated, meaning that an input that gets used in multiple places will have a fan-out corresponding to its usage frequency.

Specifying a "packing" constraint for a set of inputs requires that they be packed into a single input vector in the final circuit (note that inputs in the same vector are necessarily in different lanes). For example, a packing constraint might require that each entry of a matrix be placed in the same input vector.

After Coyote vectorizes the circuit as described above, it automatically packs the circuit inputs into vectors (while satisfying any provided packing constraints) and chooses the data layouts within these vectors. Alternatively, the programmer can choose to override this and manually provide an input layout. This is useful, for example, when composing multiple circuits, as the output layout of one determines the input layout of the next. The details of how these choices are made are discussed in Section 4.5, and the tradeoffs these annotations provide are discussed in Section 5.1.

[3]In this particular example, exchanging the positions of %3 and %4 produces semantically equivalent code that does not require the blends. However, automatically performing arithmetic rewrites such as this one is outside the scope of this work.

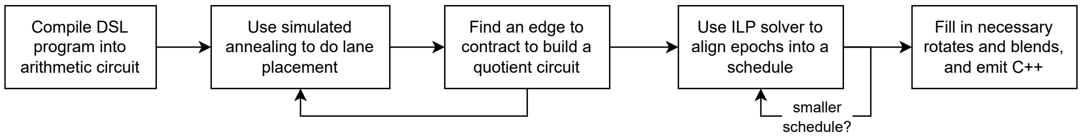
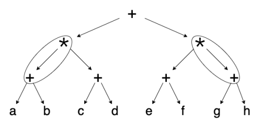
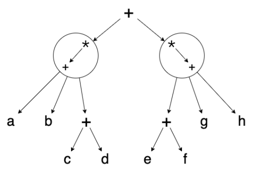

Compile DSL program into arithmetic circuit
Use simulated annealing to do lane placement
Find an edge to contract to build a quotient circuit
Use ILP solver to align epochs into a schedule
Fill in necessary rotates and blends, and emit C++
smaller schedule?

Figure 3: High-level compilation steps



(a) Circuit with subcircuits identified



(b) Quotiented circuit

```
[%1, %3] = [c, e] + [d, f]
[%0, %4] = [a, g] + [b, h]
[%2, %5] = [%0, %3] * [%1, %4]
[%6, _] = [%2, _] + [%5, _]
```

(c) Vector schedule after alignment

```
v0 = [c, e] + [d, f]
v1 = [a, g] + [b, h]
t0 = blend(v1@10, v0@01)
t1 = blend(v0@10, v1@01)
v2 = t0 * t1
s0 = v2 >> 1
v3 = v2 * s0
```

(d) Generated vector code

Figure 4: A running example of how Coyote vectorizes arbitrary arithmetic circuits

```
def dot(v1, v2):
    return sum([a * b for a, b in zip(v1, v2)])

@coyote.define_circuit(A=matrix(3, 3), b=vector(3))
def matvec_multiply(A, b):
    result = []
    for i in range(len(A)):
        result.append(dot(A[i], b))
    return result
```

Figure 5: Coyote program for multiplying a vector by a matrix

3.3 Backend

Coyote targets the BFV backend [4] for Microsoft SEAL[19]. The encryption parameters are hardcoded, and are chosen to allow for 8192 vector slots and a standard 128 bits of security.

4 DESIGN

When vectorizing arithmetic circuits with an SLP-style approach, at each step, we look at all available scalar instructions (whose source operands have all been scheduled), pick the largest set with the same operation, and schedule them together. This naïve strategy makes no guarantees about values being computed and used on the same lane; in other words, lining the computation up on incurs arbitrarily many shuffles. Unlike in normal vectorization, where applying arbitrary permutations to the lanes is relatively cheap, in FHE we are only allowed to rotate the entire vector by a fixed number of slots, and this rotation operation is expensive. Hence, the cost of bookkeeping quickly outweighs whatever benefit we might get from vectorization, making this approach not worth it.

When trying to vectorize an FHE program, we have two optimization problems to solve: instruction scheduling, and data layout. Optimizing only for instruction scheduling gives us the SLP approach: aggressively pack together isomorphic instructions without worrying about the incurred data movement overhead. Optimizing for data layout places us on the other end of the spectrum: to avoid having to do any rotates, we must place each connected component of the circuit on a single lane, precluding any vectorization and forcing us to execute everything as scalar operations.

One of our key insights is that these two problems are highly related, so we have to solve these *simultaneously* rather than independently, attempting to choose an optimal point in the tradeoff space between the two ends of the spectrum. In the following sections, we lay out the exact optimization problem as well as how we search for a solution.

4.1 Overview

The input to the compilation process is an arithmetic circuit, represented as a directed acyclic graph (DAG), where each vertex corresponds to an operation (gate) in the circuit and the leaves (vertices with no children) correspond to the inputs, and there is an arc $v_1 \to v_2$ if v_1 consumes v_2. When a particular input is used multiple times in the circuit, it can either be represented as a single vertex with an incoming arc from every gate that consumes it, or it can be *replicated* into multiple vertices which each get consumed once. This choice is expressed by the programmer in the surface language (Section 5.1).

The vector *pre-schedule*[5] is a labeled quotient of the original circuit graph, where each vertex represents a connected subgraph, and is labeled with an integer representing the lane the subgraph gets placed on, such that no two vertices at the same height are labeled by the same lane. The pre-schedule is naturally bi-graded into *epochs*, or groups of (independent) vertices at the same height which get packed together into a single sequence of vector instructions requiring no data movement, as well as *columns*, groups of vertices assigned to the same lane representing computation that happens in a single thread with no internal vectorization.

It turns out that both of our extremes from earlier can be realized in this model. Aggressively vectorizing SLP-style can be recovered by assigning a trivial subcircuit to each vertex of the quotient, and simply enumerating lanes across epochs. On the other end of the spectrum, we could instead quotient the circuit into the

[4]We could have instead chosen to use the CKKS backend, but BFV's cost model is more amenable to general vectorization. In particular, an operation we use often is "blending" slots from several vectors into one; while this is almost free in BFV, the cost of doing this in CKKS is nontrivial.

[5]This structure is referred to as a *pre-schedule* to distinguish it from the actual vector schedule, which explicitly computes an alignment for sequences of instructions at the same level.

Algorithm 1: Lane placement

```
Algorithm PlaceLanes(graph)
    lanes ← InitialPlacement(graph);
    $T \leftarrow T_0$;
    cost ← Cost(lanes, graph);
    for $i = 1 : N$ do
        $T \leftarrow T/(1+\beta T)$;
        candidate ← GenerateCandidate(lanes, graph);
        cost′ ← Cost(candidate, graph);
        if Accept(cost, cost', T) then
            lanes ← candidate;
            cost ← cost′;
    return lanes, cost
Procedure Cost(lanes, graph)
    $rotations[*] \leftarrow \emptyset$;
    foreach $(u \to v) \in graph$ do
        if $lanes[u] \neq lanes[v]$ then
            $rotations[u.epoch] \leftarrow$
              $rotations[u.epoch] \cup \{lanes[v] - lanes[u]\}$;
    $instrs[*] \leftarrow 0$;
    foreach $epoch \in graph$ do
        foreach opcode do
            $instr[opcode] \leftarrow$
              $instr[opcode] + \max_{col} count(epoch, col, opcode)$;
    return $w_R \times \sum_{ep} rotations[ep] + \sum_{op} w_{op} \times instrs[op]$
```

Algorithm 2: Computing a good circuit quotient

```
Algorithm ComputeQuotient(graph)
    lanes, cost ← PlaceLanes(graph);
    best ← lanes, graph;
    bestcost ← cost;
    pqueue ← [];
    Enqueue(pqueue, (graph, lanes), cost);
    for $i = 1 : N$ do
        graph, lanes ← Dequeue(pqueue);
        if $arc \hookleftarrow$ CrossArcs(graph) then
            candidate ←
              Condensation(ContractEdge(graph, arc));
            lanes′, cost′ ← PlaceLanes(candidate);
            Enqueue(pqueue, (candidate, lanes'), cost');
            if $cost' < bestcost$ then
                best ← lanes′, candidate;
            Enqueue(pqueue, (graph, lanes), cost);
    return best
```

discrete graph of its connected components and assign each vertex an arbitrary lane; this graph has no edges and requires no rotations, but also precludes any vectorization within connected components.

Finding a good pre-schedule then requires us to first compute a "good" quotient that trades off between these extremes, together with a lane assignment that somehow maximizes our ability to vectorize without incurring too many rotations. This is expressed in the search procedure Coyote uses when finding a vector schedule: an outer loop performs a best-first search over possible quotient graphs, and an inner loop uses simulated annealing on each quotient to find a good lane placement. The result of the search procedure is a quotient of the circuit and a lane placement for the quotient, which together minimize[6] the cost of the resulting vector schedule. The next section discusses this search procedure in more detail.

4.2 Schedule Search

Given a cost model, we use a two-layer optimization strategy to produce a schedule that has good packing properties without incurring too much data movement overhead.

Determining lane placement (Algorithm 1). The inner layer uses simulated annealing to find an optimal lane assignment for a given quotient graph. The initial assignment is the naive one given by simply enumerating the vertices at each epoch. At each step of the algorithm, we generate a candidate solution by randomly choosing two columns and a subset of the epochs in them to swap, maintaining the uniqueness condition of the schedule. If the overall cost (as described in Section 4.3) of the candidate solution is lower than the original cost, it is accepted, and used as the starting point for the next round. If the candidate solution cost is *higher* than the original cost, it is accepted with a probability that varies negatively with the difference in cost, and is generally smaller in later rounds than in earlier rounds[7]. After a fixed number of rounds have elapsed (see footnote), this algorithm returns the best solution found so far.

Computing optimal circuit quotient (Algorithm 2). The outer layer searches the space of quotients for a graph that admits a good lane placement without giving up too much vectorizability. Here, we use a priority queue to implement a simple best-first search. Each graph in the queue is assumed to already be equipped with an optimal lane placement, via the algorithm described above. At each step, a graph is dequeued, and a new candidate solution is generated by looking at its set of cross-lane arcs and choosing one to contract (removing the edge and identifying its endpoints into a single vertex). The contracted graph may not be acyclic, so we continue contracting cycles until it is (in effect computing the condensation). The candidate solution is then enqueued with its cost from the annealed lane placement. If there are more available arcs to contract, the original graph is enqueued again.

After a fixed number of rounds have elapsed, or once the queue is empty, the algorithm terminates and returns the best graph. Since each step of this algorithm involves an expensive call to the lane placement procedure, this runs for a much smaller number of rounds, usually between 150 and 200. In practice, this is enough to find highly efficient schedules.

[6]relative to the other quotients and lane placements visited in the search

[7]We use a slow cooling schedule with initial temperature $T_0 = 50$ and cooling parameter $\beta = 10^{-3}$. The probability of accepting a move that increases the cost by Δ_c is $e^{-\Delta c/T}$. The annealing is run for 20k rounds.

The next section discusses what makes one graph quotient or lane assigment "better" than another, and how these tradeoffs are quantified in Coyote's cost model.

4.3 Cost Model

The cost of a particular pre-schedule comes from two places: the number of rotations we have to perform, and the amount we have "given up" on vectorizing.

Rotations. Given a vector schedule, each *cross-lane arc* in the graph (an arc connecting vertices of different lanes) represents a rotation that must be performed to align an output from the tail of the arc to where it gets used at the head. However, determining the rotation overhead is not as simple as counting these arcs. Consider the case where instructions A and B are operands to instructions A' and B', respectively. If A and B are assigned lanes n and m, A' and B' are assigned $n + k$ and $m + k$, and A and B end up packed together in the same vector instruction, the two separate data movement operations required for the $A \longrightarrow A'$ arc and the $B \longrightarrow B'$ can actually be performed by a single rotation by k (in fact, taking advantage of this fact is the main way Coyote optimizes data layout to require fewer total rotates). To compute the *actual* number of required rotations, we instead proceed epoch-by-epoch. For each epoch, we look at all cross-lane arcs with tails in that epoch, and compute the number of columns each spans (i.e. the required rotation amount) by subtracting the lane at the tail from the lane at the head. The rotation cost for that epoch is then just the number of distinct rotation amounts. For example, if a particular epoch has five cross-lane arcs, of which three represent a rotation of -1 and two represent a rotation of 6, its rotation cost is 2. It follows that the total rotation cost of a schedule is the sum of the rotation costs of each epoch.

Vectorizability. Taking successive quotients of the circuit reduces the total number of edges, and by extension, reduces the number of rotates that might be required; however, it also precludes any vectorization within the collapsed subcircuits. To account for this, we need a way of quantifying the amount of vectorization we are "giving up" with each quotient.

Unfortunately, directly computing the opportunity cost is very messy: the amount of vectorization we give up by identifying a set of vertices is not a property local to the vertices, but rather requires us to look globally at *all possible vertices* in those epochs, to see which vectorization opportunities are no longer available after the identification. Instead, we use an estimated *schedule height* as a proxy, with the justification being that giving up a lot of vectorization generally results in taller, less efficient final schedules.

The schedule height computation also proceeds epoch-by-epoch. For each epoch, we estimate the minimum number of vector instructions after packing by taking the maximum number of each type of operation across all the subcircuits associated to the vertices in that epoch. For example, the estimated height of an epoch containing one vertex with 3 adds and 2 multiplies and another vertex with 2 adds and 4 multiplies would be 3 adds and 4 multiplies.

Overall Cost. The analysis presented above estimates the number of each type of instruction in the generated vector program. The final cost used a linear combination of all of these, with weights determined empirically by how expensive each instruction type is relative to the rest. In our implementation, we scale rotates and multiplies by 1, and addition and subtraction by 0.1.

4.4 Instruction Alignment

We align the instructions corresponding to the subcircuits in each epoch to produce a final vector schedule. It may seem like the solution to this is just sequence alignment, but aligning circuits is actually more complicated. At each step, the number of available children to align roughly doubles, meaning that the total number of subproblems to solve is exponential in the depth instead of linear. This causes the dynamic programming strategy of sequence alignment to quickly blow up.

Instead of wrangling so many subproblems, we can formulate this as an ILP. We create a variable for each scalar instruction representing its *schedule slot*, or the time at which it executes. We add constraints to require that each instruction be scheduled after all of its dependences, and also that two instructions with different operations never be scheduled at the same time. Finally, to speed up the search for a solution, we place a bound on the total length of the schedule which is iteratively tightened until the solver returns "unsatisfiable", meaning no shorter schedule could be found.

4.5 Data Layout

The circuit obtained after vectorization necessarily operates on inputs that have been "packed" into vectors. Choosing a good layout within these vectors is crucial, since a poor choice could incur many additional rotations to line operands up with where they are used. Coyote can automatically select a good layout as part of the vectorization process. An input that is only used once is placed on the lane within its vector corresponding to the unique lane where it is used, and any two inputs that are placed on the same lane by this rule are packed into separate input vectors to avoid collisions.

For inputs that are used multiple times (or inputs that are required to be packed into the same vector, e.g. elements of the same matrix), Coyote places a no-op "load" gate in the scalar circuit (so that the input is only used once, by the load gate). Two load gates are placed in the same epoch in the circuit if and only if their corresponding inputs are required to be packed together (thus ensuring that they are given different lanes). The layout for these inputs is then determined by the lanes chosen for their corresponding load gates. This determines the data layout, as each input is placed on the same lane as its corresponding load gate (Section 3.2).

5 IMPLEMENTATION

This section discusses how programmers write Coyote programs, and how the code is generated.

5.1 An eDSL and Compiler for FHE Programs

Coyote consists of an embedded DSL (eDSL) in Python that can be used to write FHE programs, shown in Figure 5. The DSL allows for arbitrary arithmetic computation over encrypted variables, and supports conditionals and loops over plaintext values. All conditionals and loops are fully evaluated and unrolled, and all function calls are fully inlined before generating the arithmetic circuit. The generated circuit is then passed to Coyote's back end, which vectorizes the computation as described in the previous sections, yielding a sequence of primitive vector operations that can be further lowered into C++ code targeting Microsoft SEAL's backend for BFV [19].

Coyote currently supports datatypes for encrypted inputs: scalar(), vector(size), and matrix(rows, cols). Inputs annotated with scalar() are free to be placed anywhere in the vector schedule; by contrast, matrix and vector inputs are always grouped together into vectors. The arithmetic circuits Coyote takes are directed acyclic graphs (DAGs) that fail to be trees exactly when values are used as inputs to multiple computations (e.g. the value a in $ab + ac$). Any such DAG can be turned into a tree by *replicating* inputs (Section 3.2) in a "reverse-CSE" process (for example, $ab + ac \to a_1 b + a_2 c$, where the value of a is supplied to both a_1 and a_2). This results in circuits with better rotation characteristics at the cost of extra computation. By default, Coyote automatically replicates all scalar inputs, and leaves all vector and matrix inputs unreplicated, but this behavior can be overridden by the programmer (note that replicating a vector or matrix input results in multiple copies of each variable all being grouped into the same vector).

Automatically choosing a data layout. While specifying a set of inputs as a vector(n) or matrix(m, n), Coyote restricts the space of available schedules to the ones that group these inputs together. However, it is still free to choose a particular layout within the vector (i.e., $a[0]$ and $a[1]$ need not be placed in adjacent lanes[8]). In practice, this allows Coyote to choose a layout that minimizes the rotations required to align inputs with where they are used. We evaluate the effectiveness of this choice in Section 6.7.

5.2 Code Generation

The algorithm in Section 4 produces a vector schedule (i.e. a lane and schedule slot for each scalar, where the schedule slot determines the order in which instructions get executed). Coyote compiles this schedule to a simple vector IR by scheduling vector instructions according to a topological sort, inserting rotations as needed. The vector ISA supports vector addition, subtraction, multiplication, and rotation, as well as a constant load instruction and a *blend* instruction. The semantics of the blend instruction are a bit subtle: it mixes lanes from multiple vector registers into the same register *while keeping all data on its original lane.* For example:

$$blend(x_1x_2x_3x_4@1010, y_1y_2y_3y_4@0101) \to [x_1y_2x_3y_4]$$

In the backend, the blend is implemented as a series of plaintext bitmasks followed by ciphertext adds.

6 EVALUATION

In this evaluation, we aim to answer the following questions:

(1) **How effective is Coyote's vectorization?** To answer this, we count the number of instructions generated in the vector code compared to the scalar code. (Section 6.2)
(2) **How much speedup does compiling with Coyote get us?** To address this, we compile several realistic benchmarks and measure the speedup of the vectorized code over scalar execution. (Section 6.3)
(3) **How well does Coyote scale up to larger kernel sizes?** To measure this, we compile a much larger circuit by vectorizing and composing its components. (Section 6.4)
(4) **How well do Coyote's schedules compare to hand optimized code?** We compare the run times of various hand-optimized benchmarks to the run time of the vector code that Coyote generates for them. (Section 6.6)
(5) **To what extent does the data layout chosen by the programmer affect Coyote's ability to vectorize?** We conduct a case study exploring various data layout schemes for a 3×3 matrix multiply, and measure the vectorization speedup for each layout. (Section 6.7)
(6) **How effective is the layout/schedule co-optimization strategy?** We track the progress of the schedule-search procedure over time for various levels of data layout optimization. (Section 6.8)
(7) **How much optimality do we sacrifice by setting synthesis timeouts?** We turn off the synthesis timeouts to guarantee optimal schedule alignments and compare the results. (Section 6.9)

6.1 Computational Kernels

To assess Coyote's ability to vectorize general applications, we use it to compile a suite of benchmarks and measure the speedup from vectorization. While there is not currently a standard benchmark suite on which to evaluate FHE-based compilers, we choose a set of benchmarks similar[9] to those used by Porcupine [5], representing a spectrum of both regular and irregular computations, as well as ones that are both sparse and dense in terms of data reuse.

The benchmarks are as follows:

1. Multiplying two matrices. We do this with 2×2 matrices (regular, little data reuse) and 3×3 matrices (regular, some substantial data reuse).
2. Vector dot product, with vector sizes of 3, 6, and 10 (all of these are regular with no data reuse).
3. 1D convolution. We do this with a vector of size 4 and a kernel of size 2, and with a vector of size 5 and a kernel of size 3. Both of these are regular and have little data reuse.
4. Point cloud distances (Given a set of points, compute the square of every pairwise Euclidean distance). We do this for 3, 4, and 5 points. These are all regular but have some substantial data reuse, especially in the 5-point case.
5. Sorting a list of size 3. This benchmark implements the sort as a "decision tree", taking as input three ciphertexts representing pairwise comparison results and six ciphertext "labels" representing possible arrangements of the sorted list. In particular, each of the three comparison results gets used in multiple branches of the tree. "grouped" here means the data layout groups the three comparisons into one vector and the six labels into another. This is irregular, and the grouped versions have data reuse.

[8]Note that a noncanonical input layout potentially means that two kernels compiled by Coyote may not be composable. We overcome this by allowing the programmer to manually provide their own layout (Section 3.2).

[9]While we do not have access to Porcupine's actual benchmarks for a direct comparison, their polynomial regression corresponds to our matrix multiply, their L2 distance correpsonds to our point cloud distance, and most of their image processing kernels are specific convolutions (i.e. plaintext kernels), while we evaluate on generic convolutions. The input sizes of our benchmarks are comparable to those of Porcupine's.

6. Finding the maximum element in a list of size 5. This benchmark takes as input five ciphertexts representing the elements of the list, and ten ciphertexts representing pairwise comparison results. Similar to the sorting benchmark, this is irregular, and the grouped versions have data reuse.

To investigate the effect of data replication we used three different replication strategies for each benchmark:

(i) *unreplicated*, where each input appears in only one input vector, and hence must be used by multiple operations.
(ii) *partially replicated* in which one of the two inputs is fully replicated—and hence each operation that requires that input gets its own copy, obviating the need for data movement—while the other is not.
(iii) *fully replicated* in which both inputs are fully replicated.

Note that for dot product, replication makes no difference as each input is used exactly once.

6.2 Costs and Effects of Vector Compilation

Table 1 shows benchmark properties, including how long each benchmark took to compile and how many operations each program had before and after vectorization. Our benchmarks ranged in size from 5 scalar instructions (size 3 dot product) to 75 instructions (5 point distances) While the number of scalar multiplies went as high as 27 (for the 3×3 matrix multiply), Coyote was almost always able to pack these into at most one or two vector instructions. The main exceptions to this rule were the highly irregular tree benchmarks, which still went from 10 scalar multiplies to between 4 and 5 vector multiplies. Another point to notice is the number of rotates. Most benchmarks required fewer than 10 rotates. However, the 5 point distance and 3×3 matrix multiply benchmarks had a very high number of rotations after vectorization, as these were the most data-dense ones—intermediate results were required by many downstream computations, incurring considerable overhead. Furthermore, the unreplicated versions of each benchmark almost always incurred more rotates than the partially or fully replicated versions, validating our hypothesis that input replication helps alleviate the rotation burden.

Table 1 also shows the ideal speedup for each benchmark, using our cost model of multiplies being 10× as expensive as adds and subtracts. Note that this modeled speedup is based on a classic work/span analysis, and hence assumes that permuting and shuffling data between vectors is free. This modeled speedup thus represents a substantial overestimate of the actual speedup that could be achieved in the program. For example, a 3x3 matrix multiply has nine 3-element dot products. The necessary 27 multiplies can all be performed in one vector operation, but the results of three multiplies need to be added to perform each dot product. Thus in reality two of the multiplies are performed in the "wrong" lane and need to be shuffled to the correct lane to complete the computation.

6.3 Speedups

While instruction counts indicate that Coyote is able to effectively find vector operations for each benchmark, we must also determine whether the actual costs of rotations and blends outweigh the vectorization benefits. Hence, we run each benchmark 50 times in scalar, and 50 times after vectorization to compute the speedup from vectorization, shown in Figure 6. We find very little variance in execution time across individual runs for any benchmark. Each benchmark has three bars representing, in order from left to right, the unreplicated, partially replicated, and fully replicated runs. The sort[3] and max[5] benchmarks have an extra green bar representing the ungrouped run (without grouping, all inputs are fully replicated no matter what). We see speedups ranging from 1.5× on the data-dense point cloud distances benchmarks to over 3.5× on the highly vectorizable matrix multiply. We also generally notice more speedup as the replication level increases, suggesting that Coyote is able to take advantage of replicated inputs to eliminate rotations from the schedule.

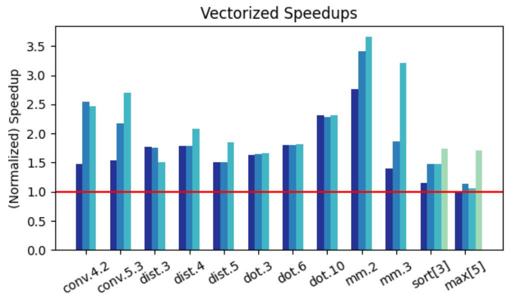


Figure 6: Speedup of vectorized code over scalar (higher is better). Left-to-right, the first three bars for each benchmark represent unreplicated, partially replicated, and fully replicated inputs, respectively. The fourth bar for the `sort[3]` and `max[5]` benchmarks represent ungrouped inputs.

While it may appear that Coyote's actual speedups are sometimes well off from the idealized speedups, this is due to the rotations and blends required outside an idealized world where vector permutation is free. For example, in the 3x3, fully-replicated matrix multiply case, Coyote generates 9 rotations to move results into place. We see, though, that in benchmarks where Coyote can generate schedules with few rotates, it does well despite the data movement costs. For example, in conv.4.2, Coyote achieves a 2.5× speedup versus an ideal speedup of 5×; and in dot.3, Coyote achieves a 1.6× speedup versus an ideal speedup of 2.7×.

6.4 Scalability

Many of the benchmarks we evaluate on have relatively small input sizes, since it is often intractable to directly apply the lane placement search procedure. However, it is possible to scale Coyote up to larger input sizes by "blocking", or vectorizing smaller kernels separately and then composing the vector programs. To investigate how well this works, we use Coyote to compile a 16×16 matrix multiply as follows. We vectorize the multiplication of a single 4×4 "block", and record the input/output layouts Coyote chooses.

The output layout of each 4×4 block is used to fix the input layout to another kernel (see Section 3.2), which takes 64 of these blocks and performs the necessary reductions to arrange them into the final 16×16 matrix multiplication. The metadata for this benchmark is shown under `mm.16` in Table 1 (the compilation time includes

Table 1: Compilation time in seconds, as well as instruction counts in the scalar (SAdd, SSub, SMul) and vector (VAdd, VSub, VMul, Rot, Blend) code, and the ideal speedup (work/span). Note that ideal speedup considers rotates and blends to be free.

Benchmark	Time (s)	SAdd + SSub	SMul	VAdd + VSub	VMul	Rot	Blend	Ideal Speedup
conv.4.2.un	97	3	6	2	1	4	4	5.73
conv.4.2.partially	80	3	6	2	1	2	1	5.73
conv.4.2.fully	71	3	6	2	1	1	2	5.73
conv.5.3.un	206	6	9	3	1	7	5	8.0
conv.5.3.partially	211	6	9	4	1	4	3	8.0
conv.5.3.fully	208	6	9	4	1	2	3	8.0
dist.3.un	228	18	9	1	1	4	6	9.82
dist.3.partially	233	18	9	1	1	4	6	9.82
dist.3.fully	226	18	9	2	2	2	7	9.82
dist.4.un	425	32	16	2	2	6	8	17.45
dist.4.partially	432	32	16	2	2	6	8	17.45
dist.4.fully	463	32	16	2	2	4	8	17.45
dist.5.un	619	50	25	2	2	13	10	27.27
dist.5.partially	629	50	25	2	2	13	10	27.27
dist.5.fully	609	50	25	2	2	9	10	27.27
dot.3.un	10	2	3	3	1	2	0	2.67
dot.3.partially	10	2	3	3	1	2	0	2.67
dot.3.fully	10	2	3	3	1	2	0	2.67
dot.6.un	159	5	6	4	1	3	2	5.0
dot.6.partially	154	5	6	4	1	3	2	5.0
dot.6.fully	156	5	6	4	1	3	2	5.0
dot.10.un	254	9	10	6	1	5	4	7.79
dot.10.partially	257	9	10	6	1	5	4	7.79
dot.10.fully	251	9	10	6	1	5	4	7.79
mm.2.un	176	4	8	2	1	3	2	7.64
mm.2.partially	171	4	8	2	1	2	1	7.64
mm.2.fully	170	4	8	2	1	1	2	7.64
mm.3.un	573	18	27	5	2	20	10	24.0
mm.3.partially	610	18	27	4	1	18	6	24.0
mm.3.fully	607	18	27	4	2	9	3	24.0
sort[3].grouped.un	238	10	10	8	4	8	6	3.24
sort[3].grouped.partially	233	10	10	8	4	4	3	3.24
sort[3].grouped.fully	231	10	10	8	4	4	3	3.24
sort[3]	139	10	10	11	5	1	1	3.24
max[5].grouped.un	880	30	30	15	11	30	18	7.33
max[5].grouped.partially	905	30	30	12	9	27	15	7.33
max[5].grouped.fully	902	30	30	19	9	24	23	7.33
max[5]	537	30	30	17	6	15	13	7.33
mm.16.blocked	7139	3840	4096	414	128	4446	872	3200

the time to compile the 4×4 block as well as the reduction circuit). After vectorizing, the blocked 16×16 matrix multiply takes 3433 seconds, compared to 4541 seconds before vectorizing, for a total 32% speedup. This shows that by composing smaller kernels, Coyote is able to scale up to vectorizing much larger circuits and still see relatively significant speedups over unvectorized code. Note that this understates Coyote's potential speedup as it does not currently attempt to vectorize separate, identical kernels together (which is a regular process so could use standard vectorization techniques).

6.5 Randomly Generated Irregular Kernels

To further investigate Coyote's ability to vectorize, even in the absence of a regular structure on the computation, we randomly

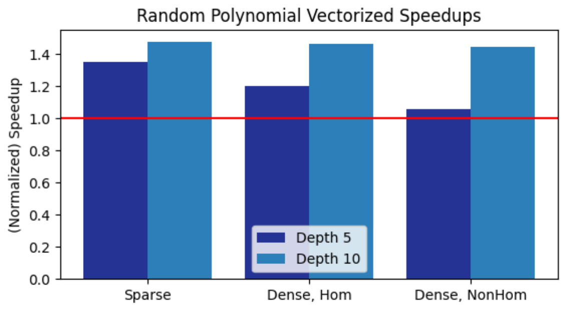


Figure 7: Speedups for random polynomials (higher is better).

generated several polynomials to evaluate as arbitrary arithmetic expression trees. The trees are generated according to three different regimes to cover different kinds of programs:

(1) *Dense, homogeneous*: The expression tree is both full and complete, and all the operations are isomorphic. In principle, this represents a best case for vectorization.
(2) *Dense, nonhomogeneous*: The expression tree is both full and complete, each operation has a 50/50 chance of being an add or a multiply. Hence, while the trees are structurally similar, the heterogeneity of operations means that vectorization opportunities are restricted.
(3) *Sparse*: Many operations have one leaf node input, the tree is not very balanced. In principle, this represents a worst case for vectorization, where Coyote must work hard to find vectorizable computation.

For each regime, we generate ten total polynomials, five with a circuit depth of 5 and five with a circuit depth of 10. Each polynomial is run 20 times in scalar and 20 times after vectorization, and we average speedups across the five polynomials in each regime before reporting. These speedups are shown in Figure 7. We see that Coyote is able to achieve speedups of up to 1.4× by vectorizing. Looking at the depth 5 dense nonhomogeneous polynomials, we found that many of them were too small and irregular to admit any profitable vectorization; in these cases, Coyote was correctly able to deduce that the scalar execution strategy was optimal rather than attempting to vectorize and incur spurious rotations. Since the generated vector code was identical to the scalar code for several of these, the average speedup is very close to 1.0.

We find that both sparse and dense homogeneous polynomials see substantial benefits from vectorization, with sparse polynomials having more speedup. This may seem surprising: dense homogeneous trees appear to be a best case scenario for vectorization, as all of the operations can be perfectly packed together. However, the key to this result is that rotations are expensive. The sparse trees have many vertices of arity 1—these operations do not require any rotations to align their input operands. In contrast, the dense trees require more rotations, canceling out the benefits from greater vectorization. This is further justification for Coyote's design decision to focus on minimizing rotation in its schedule search. In the light of this discussion, it is perhaps *un*surprising that the dense trees (requiring more rotation) with non-homogeneous operations (limiting vector packing) ultimately have the lowest speedup.

Table 2: Coyote vectorization vs. expert-written code

Benchmark	Coyote time (s)	Expert time (s)
mv.2	2.37	2.51
mv.3	3.3	3.9
mv.4	7.7	5.3
dist.3	3.5	3.2
dist.4	8.4	4.0
dist.5	15.3	5.5
dot.3	1.6	1.6
dot.6	2.9	2.2
dot.10	3.8	2.6

6.6 Comparison to Hand-Optimized Schedules

To compare Coyote's vectorized schedules to hand-optimized baselines, we compiled three kernels with Coyote: matrix/vector multiply, dot product, and point-cloud distance. Each of these kernels has a well-known expert-optimized baseline implementation, which we also implemented in Coyote's vector IR, before compiling both to C++ and measuring the time it took to run each one 50 times. The results are shown in Table 2. For smaller sizes, we see Coyote's vectorization was capable of matching or even outperforming the expert-written baselines, although on larger sizes the search space was often too big to automatically find the expert schedules. Manually inspecting the generated code shows that this was usually because the schedule Coyote generated used one or two more rotates than the baseline. In the case of the dot product, the schedules Coyote found all used the same number of rotates as the expert schedule, but occasionally incurred more blends.

6.7 Effects of Data Layout

To study the effects of different data layout choices, we vary the data layout in 3×3 matrix multiply:

Together: The matrices A and B are grouped into a single vector of 18 elements

Separate: A and B are grouped into individual vectors (this is the normal layout used in benchmarking in Figure 6)

Rows/Cols: The rows of A are grouped into three separate vectors, as are the columns of B.

Cols/Rows: The columns of A are grouped into three separate vectors, as are the rows of B.

Individual: Each of the 18 entries are grouped into their own vector (note that this is different from simply leaving them as free scalars, because this precludes Coyote from choosing to put some of them on the same vector anyway).

In each layout, all the inputs are unreplicated. Figure 8 shows the results of this case study. Interestingly, we find that grouping the matrices together yields greater speedups than keeping them separate. When multiple entries are on one vector, Coyote can arrange the elements such that rotating one automatically gives useful rotations

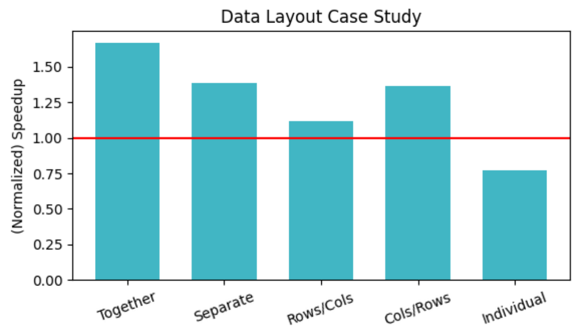


Figure 8: Speedups for the five data layout case studies (higher is better). Note that the second bar ("Separate") corresponds to the leftmost bar of mm.3 **in Figure 6.**

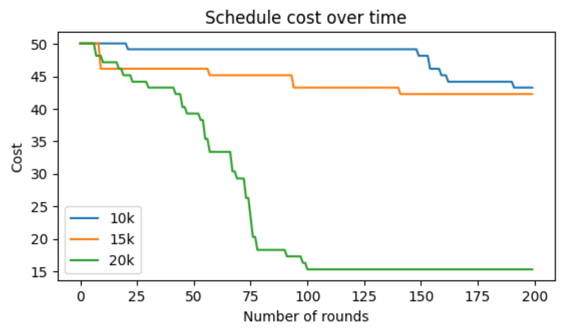


Figure 9: Schedule cost over time (lower is better) for different numbers of simulated annealing iterations for data layout per step of scheduling.

of the others. By contrast, when each entry is on a separate vector, every rotation must necessarily be done separately, so that schedule ends up with much more overhead. In particular, we find that `indiv` requires more than twice as many rotations as `together`.

6.8 Effects of Search and Co-Optimization

We would like to know how effective Coyote's schedule search is, and in particular, how good of a job it does at optimizing the data layout and schedule *together*. To test this, we compile the 5-point distances benchmark with three different levels of data layout by varying the number of iterations of simulated annealing, and record the cost of the generated schedule during each round of the search.

Figure 9 shows the cost of the vector schedule over time for various levels of data layout. The blue line depicts the schedule cost when we use 10k iterations of simulated annealing to find an optimal data layout at each step; the orange line is 15k iterations and the green line is 20k iterations. As expected, doing more iterations of simulated annealing has a large effect on the efficiency of the final schedule, since we rely on the fact that the data layout being used to guide each round of the search is close to optimal. In compiling all our benchmarks, we use 20k iterations of annealing.

Table 3: Comparison of compilation time (seconds) and vectorization speedup with vs. without synthesis timeouts.

Benchmark	Timeout		No Timeout	
	Time	Speedup	Time	Speedup
dist.5.un	619	1.70	752	1.75
dist.5.partially	629	1.73	751	1.76
dist.5.fully	609	2.08	752	2.16
conv.4.2.un	97	1.75	122	1.78
conv.4.2.partially	80	3.12	102	3.19
conv.4.2.fully	71	3.21	87	3.15
conv.5.3.un	206	1.79	260	1.80
conv.5.3.partially	211	2.48	261	2.57
conv.5.3.fully	208	3.32	254	3.30
dot.3.un	10	2.11	13	2.15
dot.3.partially	10	2.11	13	2.13
dot.3.fully	10	2.12	13	2.14
dot.6.un	159	2.51	202	2.20
dot.6.partially	154	2.51	193	2.21
dot.6.fully	156	2.51	198	2.20
dot.10.un	254	2.21	320	2.75
dot.10.partially	257	2.15	323	2.77
dot.10.fully	251	2.27	311	2.74

6.9 Optimality Tradeoffs from Timeouts

The synthesis procedure we use for generating the final (aligned) schedule from a pre-schedule uses iterative calls to an ILP solver that reduce the schedule height constraint until hitting a timeout. This is to prevent the synthesis time from blowing up, but it does come at the cost of sacrificing some optimality, since the solver might time out before finding the smallest possible schedule. In this section, we investigate the extent to which this choice matters by instead using a version of the scheduler that directly finds the minimal-height schedule with no timeout. This approach guarantees that the synthesized schedules have minimal height.[10] Table 3 summarizes the results. We see that compilation time increases with the optimal ILP, but speedups are comparable (sometimes faster, and even sometimes slower, due to different blends).

7 RELATED WORK

There are two main categories of related work. This section first discusses work on FHE vectorization, and then discusses more general approaches to plaintext vectorization

[10]Note that because the alignment algorithm cannot account for blend costs—they arise after alignment—the minimal *height* schedule may not be the minimal *cost* schedule.

7.1 Compilation and Vectorization for FHE

Prior work has been done on building vectorizing compilers for FHE applications [5, 8]. CHET [8] is a vectorizing compiler for homomorphic tensor programs that automatically selects encryption parameters, and chooses optimal data layout strategies. CHET is specifically targeted towards optimizing the dense tensor computations in neural network inference, and does not apply to a broader class of programs, especially those with irregular computations that are not so easily vectorized. Coyote makes no assumptions about the domain of the program, and can vectorize even highly irregular computations.

Porcupine [5] is a vectorizing compiler that uses a sketch-based synthesis approach to generate vectorized kernels given a reference implementation. While Porcupine is more general than CHET, it relies on programmers providing partial implementations in the form of sketches, making it less automatic than Coyote. Furthermore, Porcupine relies on the sketches to constrain the search space of rotations, while Coyote is specifically designed to reduce the number of rotations.

Gazelle [13] is a framework for secure neural network inference in FHE. While it is very optimized for a particular use case, Gazelle is not *general*: it consists of a library of highly efficient vectorized kernels that are useful in neural network applications. By contrast, Coyote can take arbitrary kernels and generate efficient vectorized code on the fly.

Lee, et. al [15] describe a general method for automatically rewriting arithmetic and boolean FHE circuits according to a cost model by learning semantically sound rewrite rules. This approach explores the space of scalar rewrites but does not directly deal with vectorization, making it orthogonal to ours: a technique like this could first be applied to an arbitrary computation to transform it into one more amenable to vectorization before applying Coyote.

One class of related work consists of compilers [1, 3, 6, 7] that automatically lift programs written in a high level DSL into optimized FHE circuits that perform the same computation. Unlike Coyote, these circuit optimizations do not include automatic vectorization. Although ACLHEMY [6] and EVA [7] *do* support ciphertext packing, they require the programmer to perform the vectorization.

7.2 General Purpose Vectorization in non-FHE Settings

Superword-Level Parallelism is a technique for automatically vectorizing programs [14]. SLP iterates over a sequence of scalar instructions and computes "vector packs," or sets of isomorphic instructions that can be packed together into vectors. Because SLP does not rely on the presence of data-parallel structures like loops, it works well even for irregular programs. When computing vector packs, SLP does not account for how expensive rotations are in FHE, leading to schedules with very high data shuffling costs.

VeGen [4] is a recent variant of SLP, introducing a notion of *lane level parallelism* that encodes which lanes are performing which computations, allowing it to reason about rotation costs when building vector packs. For example, VeGen can reason about the rotation costs to pack together operands for an instruction into a temporary vector, and can use this to decide whether or not packing those instructions is worth it. However, this reasoning only happens locally, and VeGen does not incorporate information about how instruction packing might affect later rotations.

goSLP [16] reasons about globally optimal packing, and finds lane placements that minimize data shuffling costs. However, there are assumptions baked into its cost model that make it fundamentally unsuitable for the FHE setting. goSLP frames vectorization overhead in terms of the number of *pack* and *unpack* operations incurred. For example, permuting the slots of a single vector incurs one unpack, whereas blending the contents of N vectors (without any permutation) incurs N unpacks. This cost model implicitly requires wide blends to be more expensive than arbitrary permutations, almost the opposite of FHE's cost model. In FHE, blends are almost free (instantiated as cheap plaintext multiplies and ciphertext adds) whereas a "bad" permutation can require O(n) rotates to realize. In other words, goSLP will often forego a highly profitable schedule with many blends and few rotates, and instead opt for a more conservative one. Additionally, goSLP does lane placement (permutation selection) *after* finding vector packs, creating situations like the one described in Section 1 in which the ostensibly profitable packing does not admit a good data layout. By contrast, Coyote's cooperative scheduling strategy precludes this.

There is a class of work that deals specifically with optimizing permutations in vectorized code [9, 17, 18]. Ren, et. al [18] present an algebra for reasoning about the permutation workload in SIMD programs. Eichenberger, et. al [9] develop a technique to efficiently realign memory accesses produced as a result of vectorizing a loop. Finally, Swizzle Inventor [17] automatically synthesizes efficient data movement kernels for vectorized GPU code. The primary obstacle Coyote faces in directly applying these approaches is that they tackle the data movement problem *after the kernel has been vectorized.* As we discussed earler, in the world of FHE, packing and data movemnt are problems that must be reasoned about together. There are two additional drawbacks: Eichenberger, et. al focus on aligning memory accesses in regular, data-dense loops, but this is not the setting in which Coyote operates. In the case of Swizzle Inventor, the sketches it uses to guide synthesis rely on efficiently accessing arbitrary slots of a packed vector, which is not possible in FHE without incurring significant rotation overheads.

Diospyros [22] is an equality saturation–based vectorization strategy that constructs an e-graph [21, 23] of programs that are semantically equivalent to a given specification, and then uses a custom cost model to extract an efficient vector program, together with necessary shuffles. The simplicity of the cost model it associates to various shuffles makes it unsuitable to deal with the peculiarities and inflexibility of FHE rotations.

8 CONCLUSION

This paper presented Coyote, the first vectorizing compiler for arbitrary FHE programs that considers FHE's unique cost model for data movement when vectorizing. Coyote operates at a coarser level than most vectorizers, allowing it to minimize the data movement overhead of vectorizing by only packing together sufficiently similar subexpressions. Coyote can also find optimal lane placement and data layouts that encourage more efficient rotation patterns. Coyote can automatically vectorize a large class of useful kernels, showing speedups of up to 3.5× over scalar code.

ACKNOWLEDGEMENTS

The authors appreciate the feedback from anonymous reviewers from PLDI 2022, OOPSLA 2022 and ASPLOS 2023 that have improved the paper. This work was partially supported by NSF grants CCF-1908504 and CCF-1919197, as well as Cisco.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains everything necessary to replicate the results of this paper, including:

- An implementation of the compiler described in the paper
- A backend test harness for profiling the vectorized code Coyote generates
- Implementations of all the benchmarks used in the evaluation
- Various scripts necessary to automate the process of compiling, running, and collecting data from the benchmarks.

Note that there are two experiments omitted from the artifact, as they require nontrivial manual effort to set up and run. These are the `mm.16.blocked` benchmark described in Section 6.4, and Figure 9

A.2 Artifact Check-List (Meta-Information)

- **Compilation:** Translates a python program into an arithmetic circuit, vectorizes it, and generates C++ FHE code
- **Transformations:** Loop unrolling, function inlining, circuit vectorization
- **Experiments:** Compiling real-world benchmarks, compiling randomly generated polynomial programs, experimenting with data layouts
- **How much disk space required (approximately)?:** 200MB
- **How much time is needed to complete experiments (approximately)?:** 45 minutes - 1 hour for the small version, up to several hours for running all the benchmarks
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Archived:** 10.5281/zenodo.7591603

A.3 Description

A.3.1 How to access. The Coyote compiler can be accessed at https://github.com/raghav198/coyote

A.3.2 Hardware Dependencies. No specialized hardware is required to use Coyote, beyond whatever may be necessary to efficiently run z3 and SEAL.

A.3.3 Software Dependencies.

- The `coyote` compiler is implemented in Python 3.10 and uses the networkx and z3 modules for its analysis
- The test harness backend is written in C++ and uses version 3.7 of the Microsoft SEAL library for its FHE implementation
- The test harness uses cmake for its build system

A.4 Installation

The Dockerfile provided with the artifact automatically builds and installs all dependencies of Coyote. To build and run the Docker image, run the following commands from the directory containing the Dockerfile:

```
$ docker build -t coyote .
$ docker run -it coyote bash
```

A.5 Experiment Workflow

This section describes a workflow to reproduce a subset of the results in the paper. We've recorded the approximate time it takes to complete each step inside the Docker image on a 2020 M1 MacBook Air. Lets start by building all the `small` benchmarks (all replication sorts for `conv.4.2`, `mm.2`, `dot.3`, `dot.6`, and `dot.10`, as well as the ungrouped `sort[3]`).

```
$ python3 compile_benchmarks.py --preset small
```

We can also build the data layout case study from Section 4.5 of the paper, although note that these circuits are considerably larger, so compiling them will take some time:

```
$ python3 compile_benchmarks.py --preset layout
```

Lets also build some of the polynomial trees; in particular, we'll build two of the depth 5 trees in each of the three regimes.

```
$ python3 polynomial_benchmarks.py -d 5 -r \
"100-100" "100-50" "50-50" -i 2
```

We can see, for example, some of the Coyote vector IR:

```
$ cat sort_3/vec
```

and the corresponding generated vector C++ code:

```
$ cat bfv_backend/coyote_out/sort_3/vector.cpp
```

To build *all* the benchmarks from the paper (small, medium, and large, as well as the layouts and the random polynomials), run the following instead of following the above steps:

```
$ python3 coyote_compile.py benchmarks.py -c "*"
$ python3 polynomial_benchmarks.py -d 5 10 -r \
    "100-100" "100-50" "50-50" -i 5
```

However, this is not recommended and will take several hours to complete, as several of the circuits being compiled are quite large.

Now, we need to compile all the C++ code and collect data. Although we used 50 runs and 50 iterations in the paper, lets only use 10 of each to make this go faster:

```
$ python3 build_and_run_all.py \
        --runs 10 --iters 10
```

You should see some CMake output followed by the encryption and run times for both scalar and vector versions of each circuit. Note that this script will not re-run benchmarks that already have corresponding CSV files in `bfv_backend/csvs/`. Once this is finished running, we can look at one of the generated CSV files:

```
$ cat bfv_backend/csvs/sort/sort_3.csv
```

Now that we've collected all the data for these benchmarks, we can generate the graphs:

```
$ python3 figures.py
```

This will generate three plots: vector_speedups.png, case_study.png, and trees.png. To view these, either attach to the running Docker container (e.g. using VS Code), or copy the files to your host machine:

```
$ docker cp $(docker ps -q):/home/artifact/graphs/ .
```

Compiling all the small benchmarks takes about 13 minutes, generating and compiling the random polynomial benchmarks takes about 5 minutes, compiling the data layout case study takes about 15 minutes, and building and running all the benchmarks takes about 15 minutes.

A.6 Evaluation and Expected Results

After running through the workflow described above, you should have generated three plots, each of which replicates part of the experiments in this paper:

- `vector_speedups.png` corresponds to Figure 6
- `trees.png` corresponds to Figure 7
- `case_study.png` corresponds to Figure 8

Note that the generated graphs may not contain all the experiments found in the paper (for example, not all the benchmarks in Figure 6 are built in the above workflow, and neither are the depth 10 trees in Figure 7, as these take a long time to compile). However, the speedups should resemble those in the corresponding figures.

A.7 Experiment Customization

A.7.1 Writing a Coyote program. Coyote is a DSL embedded in Python, so Coyote programs are just Python functions. To tag a function as a circuit for Coyote to compile, first get an instance of the Coyote compiler:

```
from coyote import *
coyote = coyote_compiler()
```

Next, use the compiler to annotate your function with input types:

```
@coyote.define_circuit(A=matrix(3, 3), B=matrix(3, 3))
def matrix_multiply(A, B):
    ...
```

For a full discussion of the available types and their compile-time semantics, see Section 5.1 Finally, use the build script `coyote_compile.py` to invoke the Coyote compiler on the Python file in which this code is saved:

```
python3 coyote_compile.py circuits.py -c \
    matrix_multiply
```

A.7.2 Invoking the Compiler. The Coyote compiler can be invoked from the command line via `coyote_compile.py`. The example invocation above does the following:

(1) It parses `circuits.py` and loads a list of all circuits defined in that file
(2) It uses Coyote to compile the specified `matrix_multiply` circuit
(3) It creates a directory called `matrix_multiply` and saves intermediate scalar and vector code into that directory
(4) It lowers the intermediate code into C++ and saves it in `bfv_backend`

The script expects the name of a Python file that defines one or more circuits (as described above), and then takes a number of command-line parameters:

- `-l`, `-list`: Lists all the circuits defined in the file and exit, does not actually compile anything
- `-c`, `-circuits`: Load the specified circuits from the file and compile them into C++
- `-o`, `-output-dir`: Specify the directory into which to place the generated intermediate code (defaults to the directory from which coyote_compile.py is invoked)
- `-backend-dir`: Specify the directory containing the test harness backend (defaults to bfv_backend/)
- `-no-cpp`: Stops after generated the intermediate code and doesn't generate C++
- `-just-cpp`: Uses pregenerated intermediate code to generate C++ instead of recompiling the circuit; this fails if it can't find the intermediate code under [output-dir]/[circuit-name]/

A.7.3 Running the Test Harness. The backend test harness comes with a CMake file that automatically builds binaries for everything under `coyote_out/`. The generated binaries perform a number of runs, where each run consists of executing the scalar and vectorized circuits on random encrypted inputs for a number of iterations and then outputting the total time each version (scalar and vector) took to encrypt, as well as run. All these outputs are then saved into a csv file with the same name as the circuit (e.g. running the binary generated from the example above would create a file called `matrix_multiply.csv`).

The number of runs and iterations default to 50 each (as these are the values used in this paper), but are configurable via cmake. An example invocation that uses 10 runs with 10 iterations each is as follows:

```
$ mkdir bfv_backend/build
$ cd bfv_backend/build
$ cmake .. -DRUNS=10 -DITERATIONS=10
$ make -j16
```

REFERENCES

[1] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. 2019. RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (London, United Kingdom) *(WAHC'19)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/3338469.3358945

[2] Zvika Brakerski, Craig Gentry, and Shai Halevi. 2012. Packed Ciphertexts in LWE-based Homomorphic Encryption.

[3] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: A Compilation Chain for Privacy Preserving Applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing* (Singapore, Republic of Singapore) *(SCC '15)*. Association for Computing Machinery, New York, NY, USA, 13–19. https://doi.org/10.1145/2732516.2732520

[4] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[5] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 375–389. https://doi.org/10.1145/3453483.3454050

[6] Eric Crockett, Chris Peikert, and Chad Sharp. 2018. ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1020–1037. https://doi.org/10.1145/3243734.3243828

[7] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*

(London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 546–561. https://doi.org/10.1145/3385412.3386023

[8] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 142–156. https://doi.org/10.1145/3314221.3314628

[9] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 82–93. https://doi.org/10.1145/996841.996853

[10] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[11] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme.* Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) Boneh, Dan. AAI3382729.

[12] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106. https://ia.cr/2014/106.

[13] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1651–1669. https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[14] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[15] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 503–518. https://doi.org/10.1145/3385412.3385996

[16] Charith Mendis and Saman Amarasinghe. 2018. GoSLP: Globally Optimized Superword Level Parallelism Framework. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 110 (oct 2018), 28 pages. https://doi.org/10.1145/3276480

[17] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3297858.3304059

[18] Gang Ren, Peng Wu, and David Padua. 2006. Optimizing Data Permutations for SIMD Devices. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 118–131. https://doi.org/10.1145/1133981.1133996

[19] SEAL 2021. Microsoft SEAL (release 3.7). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[20] Nigel Smart and Frederik Vercauteren. 2011. Fully homomorphic SIMD operations. *IACR Cryptology ePrint Archive* 2011 (01 2011), 133. https://doi.org/10.1007/s10623-012-9720-4

[21] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. *SIGPLAN Not.* 44, 1 (jan 2009), 264–276. https://doi.org/10.1145/1594834.1480915

[22] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. *Vectorization for Digital Signal Processors via Equality Saturation.* Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[23] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

Received 2022-10-20; accepted 2023-01-19