Biscotti: An Approach to Parallel Scheduling for Vectorized Encrypted Arithmetic Circuits

Sreevickrant Sreekanth*
ssrekan@purdue.edu
Purdue University

Dulani Wijayarathne*
dwijayar@purdue.edu
Purdue University

Raghav Malik
malik22@purdue.edu
Purdue University

Milind Kulkarni
milind@purdue.edu
Purdue University

ABSTRACT

Fully Homomorphic Encryption (FHE) enables secure computation on encrypted data, ensuring privacy in various applications. However, FHE's practicality is constrained by the substantial overhead of encrypted computation. This overhead can be partially mitigated by using vectorization strategies to optimize FHE computations. The unique semantics of FHE computations make vectorizing arbitrary applications challenging, so recent research focuses on applying synthesis-based techniques, which have the drawback of failing to scale well to vectorizing large programs.

This paper proposes, Biscotti, a compiler pass that integrates with existing synthesis-based vectorization approaches by breaking programs down into more manageable pieces, synthesizing vector schedules for each piece, and then composing these into a schedule for the entire program. We demonstrate on a variety of common benchmarks that Biscotti's approach not only improves compilation times, but also results in more efficient schedules overall.

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) refers to a class of encryption schemes that support performing computations directly on ciphertexts without needing to decrypt them first. FHE is crucial for applications such as secure multiparty computation and secure machine learning, but the extreme inefficiency of encrypted computation presents a barrier to its widespread adoption: even state-of-the-art FHE implementations incur an overhead of several orders of magnitude when compared to carrying out the same computation over unencrypted data [9, 17].

1.1 Vectorizing in FHE

The formulation of many FHE schemes enables programmers to alleviate some of the overheads of encrypted computation via an optimization known as *ciphertext packing*. Schemes that support ciphertext packing allow multiple plaintext values to be encrypted into a single ciphertext *vector*, so that homomorphic operations on the ciphertext vector occur elementwise on the underlying plaintexts (Section 2.2). These schemes therefore allow programmers to vectorize their FHE computations, by, for example, replacing a sequence of n independent multiplications with a single vectorized operation; this results in fewer homomorphic operations overall, and hence, better performance.

Manually vectorizing arbitrary programs is difficult, and FHE vectors have some peculiar semantics that make vectorization particularly tedious. In particular, shuffling data between vector slots (hereafter called *lanes*) can only be done via an expensive operation called *rotation*; hence, minimizing rotations is crucial to achieving efficient vector schedules. Vectorizing FHE compilers can be broadly classified by how they address this issue. On one side of the spectrum, tools like Coyote [13] and Porcupine [4] frame vectorization as a search problem, and use synthesis techniques to find efficient rotation patterns for arbitrary computations. While these techniques tend to work quite well for programs with a small number of instructions, they often fail to scale up to larger programs with a more massive search space.

On the other hand, compilers such as HECO [18] and CHET [7] focus on vectorizing programs that require very few rotations in the first place, often relying on language constructs like arrays and tensors to determine when this is possible. They can therefore scale up to much larger kernels, but this comes at the cost of placing restrictions on the class of programs they can vectorize, making them significantly less general than the aforementioned search-based techniques.

1.2 The Middle Ground

The FHE vectorizers described so far either *generalize* or *scale*; in this paper, we present Biscotti, a vectorizer that does *both* by achieving a "happy medium" between the two extremes. The key insight that enables this is that large, complex programs can usually be *decomposed* into computations that are smaller, and therefore, crucially, *easier to vectorize*.

As an example, consider the 8-element convolution shown in Figure 1, which can be decomposed into two smaller convolutions over 5 and 6 elements. While the search timeouts cause Coyote to generate suboptimal schedules for the full 8-element convolution, it is able to quickly find much better schedules for each subprogram. In fact, by running the vectorized subprograms *in parallel*, we can achieve a much better schedule overall!

*Equal contribution

This forms the core of Biscotti's vectorization procedure. In particular, Biscotti starts *decomposing* a large computation into smaller subprograms. It uses Coyote to quickly synthesize efficient vector schedules for each subprogram, and then *composes* each vectorized subprogram back into a much more efficient schedule for the original computation.

1.3 Contributions

The specific contributions we make in this paper are:

(1) An algorithm to decompose and compose subprograms together to enable parallel execution.
(2) Modifications to the embedded DSL (eDSL) in Coyote for loop tiling.
(3) Extension of the said algorithm for parallel execution of bounded-depth recursion and function compositions.

We tested Biscotti to compile three computation kernels (point clound distances, matrix multiplicaitions and 1D convolution) and compared the compilation and runtimes with with Coyote compiled programs. We also investigate the effects of choosing subprograms of varying granularity. We find that Biscotti, yields better compilation times, more efficient schedules and better runtimes.

2 BACKGROUND

2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) is an encryption paradigm that enables the computation of arbitrary functions on encrypted data without requiring a decryption key. This capability has a wide array of applications, including facilitating private search queries, conducting encrypted data searches, and enabling efficient and secure multiparty computation [8]. We use the the Brakerski/Fan-Vercauteren (BFV) cryptosystem, which is founded on the Ring Learning With Errors (RLWE) problem [1].

Limitations. First, homomorphic arithmetic operations applied to ciphertexts often exhibit performance that is orders of magnitude slower compared to their plain text equivalents. This slowdown is even more pronounced when the ciphertext size increases, which can happen in larger circuits or with higher security parameters. Consequently, many practical applications that utilize fully homomorphic encryption (FHE) circuits face a notable slowdown in execution, rendering them impractically slow. Overcoming these performance challenges requires substantial expertise and experience in crafting FHE programs. Furthermore, the security assurances provided by FHE necessitate that all operations are data independent. While certain forms of conditional logic can be emulated, every potential branch must be evaluated, leading to considerable degradation in performance.

2.2 Vectorization

The mathematical formulation of Fully Homomorphic Encryption (FHE) allows for ciphertext packing, a technique where scalar computations are extended to operate over packed vectors. Ciphertext packing helps enhance the overall runtime complexity and effectively establishes a Single Instruction, Multiple Data (SIMD) architecture. Classical SIMD-methods rely on loop vectorization, wherein a data-parallel loop is unrolled by a fixed number of iterations to produce a set of isomorphic instructions which are then vectorized together. In contrast, Superword-Level Parallelism (SLP), focuses on vectorizing arbitrary, non-loop based code by identifying groups of isomorphic independent instructions and scheduling them into vectors [10, 14].

Vectorization in FHE. The BFV encryption scheme is a specific instantiation of FHE which permits packing many plaintext elements into a ciphertext [1]. Homomorphic operations correspond to element-wise operations on the underlying vectors. The BFV scheme also supports certain operations that allow vector slots (hereafter referred to as *lanes*) to be permuted cyclically (hereafter referred to as *rotations*). This style of vectorization has a few attributes that differentiate it from normal vectorization:

(1) Vector slots are on the order of thousands compared to hardware vectors that are only a few slots wide. Therefore, it becomes essential to use as many vector slots as possibles to ameliorate loss in performance.
(2) Absence of indexing prevents accessing a value directly from a slot of a ciphertext vector.
(3) Data movement between slots in a vector is enabled by rotating the vector. This puts a heightened emphasis on assigning the correct vector lanes when packing instructions, because realizing arbitrary permutations is computationally expensive. It is often more prudent to give up vectorization, to avoid incurring costly rotations [13].

2.3 Coyote

Coyote is an FHE-aware vectorizing compiler that targets programs that do not have regular structure [13]. Given an arithmetic circuit, Coyote produces a large graph enumerating potential vector schedules, and then searches this space to find a good one. However, the size of the graph is roughly exponential in the size of the program, so this search procedure quickly becomes intractable, even for moderately sized programs. Importantly, this means that for larger programs, the search often times out and produces sub optimal schedules.

3 OVERVIEW

Biscotti is built as an optimization on top of Coyote (Section 2.3). Recall that while Coyote can synthesize good vector schedules for smaller programs, its search strategy quickly becomes intractable for larger programs. Biscotti is an optimization built on top of Coyote that addresses this limitation by noticing that large programs can often be broken down into smaller, more manageable subprograms. We can realize the patterns of decomposing a larger program through the following methods:

(1) *Tiling*: We can decompose a program's iteration space into smaller, repetitive 'tiles,' each of which can be independently compiled. This loop transformation technique is used by compilers to facilitate the creation of blocked algorithms [19]. We coalesce isomorphic operations in an iteration space, i.e. split the index set into well-structured iteration subsets, generate schedules for each element in the subset, and parallelize them.

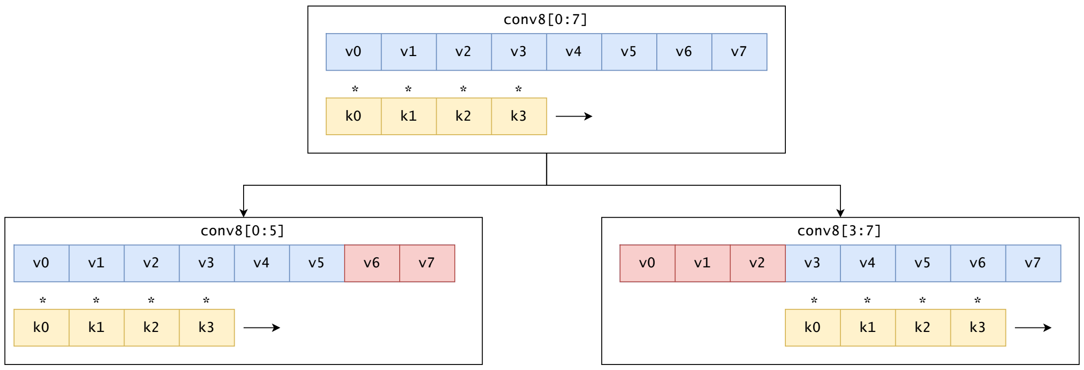


Figure 1: Convolution of an 8-element `signal` vector with a 4-element `kernel` vector decomposed to two subprograms.

(2) *Recursion*: After generating the call tree of a bounded-depth recursive function to a desired depth, we can isolate symmetric sub-graphs and generate a vector schedules for the sub-graphs independently. Following compilation, we parallelize the schedules, followed by the addition of necessary *reduction* steps as specified by the call tree to avail the final outcome of the program.

(3) *Function Compositions*: Consider a function f composed of f_1 and f_2, denoted by $f = f_1 \circ f_2$. Vector schedules for each function f_1 and f_2 are compiled independently and a composite schedule is generated with calls to functions f_1 and f_2.

Biscotti uses Coyote to vectorize subprograms generated by *tiling* and *composes* the schedules back into a vector schedule for the large program. We discuss the other two techniques, *recursion* and *function composition*, in more detail in Section 7. We explain the overall workflow of the Biscotti by using the running example of convolving an 8-element `signal` vector with a 4-element `kernel` vector, as illustrated in Figure 1.

Decomposition. Biscotti begins by breaking down a large program into smaller subprograms by using loop tiling as detailed in Section 5.

```python
# Original loop structure
def conv8(signal, kernel):
    for i in range(5):
        sum = 0
        for j in range(4):
            sum += signal[i] * kernel[j]
        output[i] = sum
    return output
```

As shown in the original loop structure above, the initial implementation iterates over an 8-element `signal` vector.

```python
# Loop tiling with a factor of 2
def conv8_loop1(signal, kernel):
    for i in range(3):
        sum = 0
        for j in range(4):
            sum += signal[i] * kernel[j]
        output[i] = sum
    return output

def conv8_loop2(signal, kernel)
    for i in range(3, 5):
        sum = 0
        for j in range(4):
            sum += signal[i] * kernel[j]
        output[i] = sum
    return output
```

We modify the original loop structure by applying a tiling factor of 2, which segments the iteration space into 2 smaller blocks. This restructuring does not directly speed up the loops when run separately; instead, it enhances their suitability for vectorization by tools such as Coyote. These smaller, more manageable blocks significantly increase the chances for Coyote to identify highly efficient schedules for these segmented iteration spaces, or subprograms. The resulting schedules generated by Coyote is demonstrated in Figure 2.

Composition. We retrieve the schedules produced in *decomposition*, align similar instructions as illustrated in Figure 3 and compose them together through *interleaving* to enable parallel execution of the subprograms. We discuss composition and interleaving in more detail in Section 4.2.

4 DESIGN

The Biscotti framework employs a two-step method involving Decomposition and Composition. It begins by breaking down loops according to a programmer defined tiling factor and target loop, as outlined in Section 5. In the composition phase, we ensure the efficiency gains from Coyote are retained by correctly merging the optimized schedules. Incorrect composition can result in additional computation and unnecessary instructions, effectively multiplying the workload. To prevent this, we propose an interleaving method for schedule integration, discussed further in Section 4.2.

4.1 Decomposition

We primarily focus on loop tiling from the decomposition strategies outlined in Section 3. It is relatively straightforward to extend these ideas to the other strategies as discussed in Section 7.

Biscotti allows the programmer to annotate the loops they wish to tile by a tiling factor. Biscotti segments the iteration space of the specified loop, by modifying the specified loop indices based on the tiling factor. We refer to these segmented iteration spaces as *subprograms*. Section 5 delves deeper into the specifics of how programmers can annotate the loops to tile. This process effectively divides the program into independent subprograms that can be executed simultaneously. The choice of the tiling factor plays a role in the optimization of the compilation and run times. We discuss the implications of choosing different tiling factors in Section 6.4.

In the running example in Figure 1 the `signal` vector of size 8 with a `kernel` of size 4, the outer loop is broken down into two tiles by modifying the loop indices into tiles to `{[0:3], [3:5]}`. Coyote generates vector schedules for the two tiles in parallel. Figures 2 show the Coyote generated schedules for the loop tiles.

4.2 Composition

4.2.1 Operation Sequencing. We cannot guarantee that Coyote will generate isomorphic schedules even if two different subprograms have the same iteration spaces because Coyote's search procedures for the decomposed subprograms run independently; they may converge on different solutions. Therefore, Biscotti must align vector instructions with the same operations together before schedule composition. The schedules in Figures 2 from the convolution kernel have varying vector instructions which must be aligned before composing the instructions together. We formalize this process through the computation of a sequence alignment between the operations across all the different sub-schedules by utilizing the Needleman-Wunsch algorithm [15]. Figure 3 illustrates the alignment of operations from the two schedules in Figure 2. It should be noted that in the alignment depicted in Figure 3a, two instructions from schedule 2 remain unaligned. Consequently, these instructions are aligned with no-operations (no-ops) in the finalized schedule.

4.2.2 Executing the Interleave. Before interleaving process of the schedules, register numbers must be renumbered to ensure program correctness and to avoid conflicts between different schedules during composition.

For each aligned vector instruction in the schedules, we interleave the lanes by alternating between lanes while maintaining their original sequence. We combine the lanes from different vector schedules into a unified instruction, where the order of elements from each original sequence is preserved. Figures 4a and 4b shows two rotations instructions. Figure 4d shows the interleaved results of the instruction wherein the first lane from `instruction 1` is scheduled in `lane 0`, and the first lane from `instruction 2` is scheduled in `lane 1`. We repeat this until all the aligned vector instructions sets have been composed together.

4.2.3 Side by Side vs Interleaving. We choose interleaving over side-by-side placement to parallelize subprogram vector schedules. This choice is motivated by the efficiency of interleaving in handling rotation operations (Section 2.2), a key idea of optimization in the Coyote compiler. When comparing to a side-by-side arrangement, as shown in Figure 4c, trying to align data slots from each subprogram in an interleaved format can lead to an excessive number of rotations and multiple vector maskings to achieve the same number of rotations and alignment in lanes as represented in Figures 4a and 4b. Given how expensive each rotation is, attempting to parallelize subprogram schedules this way counteracts the optimization benefits provided by Coyote, making it less effective to produce an optimized final schedule.

In contrast, interleaving operations from one schedule with another helps to reduce the overall number of rotations needed to achieve a specific alignment in the interleaved schedule. For instance, a single leftward rotation in both subprogram schedules, as illustrated in 4d, can be achieved by preserving the same amount of rotations in Figures 4a and 4b. This method aligns with Coyote's objective to decrease rotational operations, optimizing the efficiency of vector scheduling.

5 IMPLEMENTATION

This section discusses how programmers can use Biscotti, to write Coyote programs, and how the code is generated.

Coyote provides an embedded Domain-Specific Language (eDSL) within Python, designed for writing FHE programs. This DSL empowers users to perform arbitrary arithmetic computations over encrypted variables, leveraging conditionals and loops over plaintext values to express complex algorithms. Before generating the arithmetic circuit, all conditionals and loops are fully evaluated and unrolled, with all function calls being fully inlined. This process ensures a direct translation of high-level constructs into a form suitable for encrypted computation.

We integrate Biscotti into the eDSL, allowing users to specify tiling as part of the function declaration. This is achieved through an enhanced decorator mechanism, where the user can indicate the desired tiling for loops within the function. Biscotti then automatically segments the iteration space by manipulating the Abstract Syntax Tree (AST) of the specified loops according to the tiling parameters, ensuring that each tile is fully unrolled and inlined in preparation for circuit generation.

```
@coyote.define(signal=vector(8), kernel=vector(4),
↪  tile=4, loop_target=i)
def conv8(signal, kernel):
    for i in range(5):
        sum = 0
        for j in range(4):
            sum += signal[i] * kernel[j]
        output[i] = sum
```

Once the arithmetic circuit is generated, reflecting both the computation logic and the applied loop tiling optimizations, Coyote's backend takes over. It vectorizes the tiles in parallel, translating the high-level circuit into a sequence of primitive vector operations.

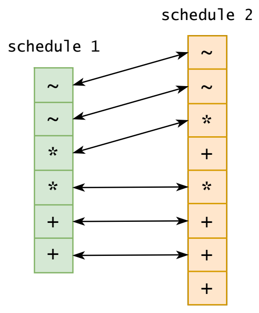
```
[%0, %19, %3, %27, %7, %10] = [v0, v4, v1, v5, v2, v3]
[_, _, %8, %4, %11, %1]     = [_, _, k2, k1, k3, k0]
[%20, %5, %28, %16, %12, %24] = [%19, %3, %27, %7, %10, %10] * [%11, %4, %11, %4, %11, %4]
[%18, %2, %26, %15, %9, %23]  = [%10, %0, %19, %3, %7, %7]   * [%8, %1, %8, %1, %8, %1]
[%21, %6, %29, %17, %13, %25] = [%18, %2, %26, %15, %9, %23] + [%20, %5, %28, %16, %12, %24]
[%22, %14, %30, _, _, _] = [%17, %6, %25, _, _, _] + [%21, %13, %29, _, _, _]
```

(a) Schedule for conv[0:5] (schedule 1)

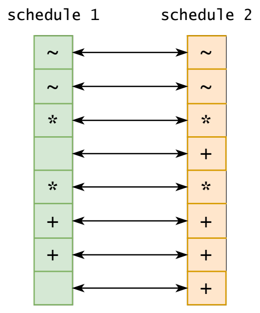
```
[_, %19, %10, _, %7, _, %3, %0]       = [_, v7, v6, _, v5, _, v4, v3]
[%11, _, %8, _, %4, _, %1, _]         = [k3, _, k2, _, k1, _, k0, _]
[%20, %12, _, %9, %6, %5, %15, %2] = [%19, %10, _, %7, %7, %3, %3, %0] * [%11, %11, _, %8, %4, %4, %1, %1]
[_, _, _, _, _, _, %17, %6]           = [_, _, _, _, _, _, %15, %2]   +       [_, _, _, _, _, _, %16, %5]
[_, _, %18, _, _, _, _, _]            = [_, _, %10, _, _, _, _, _]    *       [_, _, %8, _, _, _, _, _]
[_, _, %21, _, _, _, _, _]            = [_, _, %18, _, _, _, _, _]    +       [_, _, %20, _, _, _, _, _]
[_, _, %22, %13, _, _, _, _]          = [_, _, %7, %9, _, _, _, _]    +       [_, _, %21, %12, _, _, _, _]
[_, _, _, %14, _, _, _, _]            = [_, _, _, %6, _, _, _, _]     +       [_, _, _, %13, _, _, _, _]
```

(b) Schedule for conv[3:7] (schedule 2)

Figure 2: Coyote generated schedules after decomposition through tiling

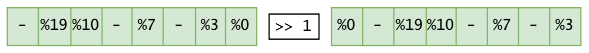


(a) Original alignment

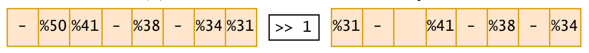


(b) Post-alignment

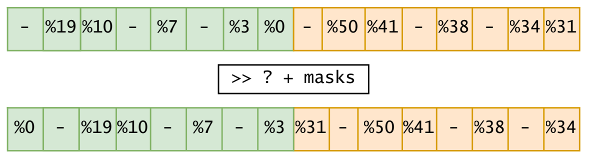
Figure 3: Vector Operations Sequence Alignment: Original vs. Post-Alignment



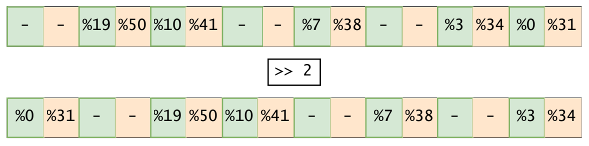
(a) Instruction 1 - Rotation by 1



(b) Instruction 2 - Rotation by 1



(c) Side by Side Scheduling where the concatenated vectors must rotated at varying degrees and then merged through masking multiple variations of the same vector to achieve the targeted outcome.



(d) Interleaved Scheduling where rotations are preserved from the instruction being composed.

Figure 4: Scheduling vectors Side by Side vs. Interleaving

Biscotti, then uses the generated vector schedules and composes all of them together in two passes: (1) Sequence Alignment and (2) Interleaving. This vectorized form is then further lowered into C++ code, targeting the Microsoft SEAL backend for the BFV scheme [17]. The incorporation of loop tiling into this workflow optimizes the underlying arithmetic circuits, making them more efficient and suitable for the computational demands of FHE, thereby enhancing the overall performance of Coyote-generated programs.

6 EVALUATION

In this evaluation, we address the following questions:

(1) **How much faster are Biscotti's compile times compared to Coyote's compile times?** We compile the benchmarks described in Section 6.1 using both Biscotti and Coyote, and compare how long each benchmark takes to compile (Figure 5). Biscotti's compilation strategy for each benchmark is described in Section 6.2.
(2) **How efficient is the code that Biscotti generates compared to Coyote?** We compile the benchmarks in Section 6.1 using both Biscotti and Coyote, and compare the runtimes across 10 iterations.
(3) **How does the size of the tiles influence the compilation time and execution time of the code generated by Biscotti?** We compile the benchmarks in Section 6.4 with varying tiling factors and compare the compile times and runtimes.

6.1 Benchmarks

We answer questions 1 and 2 using the following benchmarks.

(1) Point Cloud Distance between 2 vectors of size 10 with a tiling factor of 2.

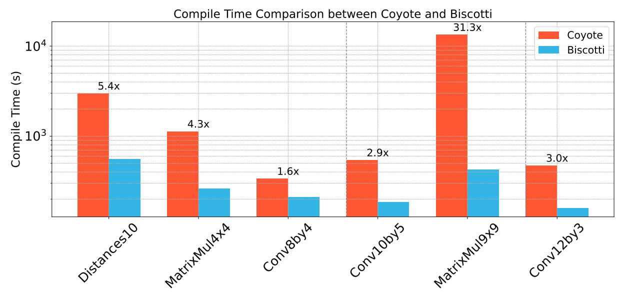


Figure 5: Compilation time Coyote vs Biscotti

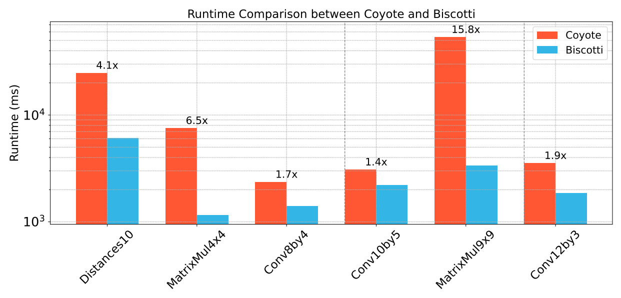


Figure 6: Runtime Coyote vs Biscotti

(2) Matrix multiplication of a 4×4 matrix with a tiling factor of 2.
(3) Convolution with a vector of size 8, a kernel size of 4, with a tiling factor of 2.
(4) Convolution with a vector of size 10, a kernel size of 5, with a tiling factor of 3.
(5) Matrix multiplication of a 9×9 matrix, with a tiling factor of 3 .
(6) Convolution with a vector size of 12, a kernel size of 3, with a tiling factor of 4.

6.2 Compile Time Comparison of Biscotti with Coyote

As demonstrated in Figure 5, Biscotti offers a substantial reduction in compilation times compared to Coyote, achieving up to a 31.3x speedup, particularly noted in Benchmark 5. These results validate the efficiency of Biscotti's loop transformation technique, which mitigates the challenges of large arithmetic circuits by decomposing large programs into more manageable subprograms. Consequently, Biscotti enables faster identification of vector schedules for these subprograms, leading to reduced compilation times. It compiles these subprograms concurrently to generate vector schedules for each subprogram via Coyote. The integration of these individual schedules into a comprehensive final schedule, as outlined in Section 4 has a negligible impact on the total compilation time.

6.3 Runtime Comparison of Biscotti with Coyote

Figure 6 illustrates Biscotti's advantage over Coyote in runtime performance. By generating subprograms from less complex arithmetic circuits, Biscotti reduces the size of the space Coyote has to search, making it more likely to find a more optimal vector schedule. The performance improvements are consistent across all benchmarks, with each showing a minimum of 1.4x speedup. The significant 15.8x speedup demonstrated in Benchmark 5 highlight Biscotti's capability to overcome the challenges that Coyote faces with large circuits. We also support our assertion that Biscotti generates fewer vector instructions overall compared to Coyote from Table 1 by comparing the number of vector instructions generated by the two systems. For instance, Benchmark 5 has the highest number of vector instructions generated by Coyote, totaling 426 while Biscotti

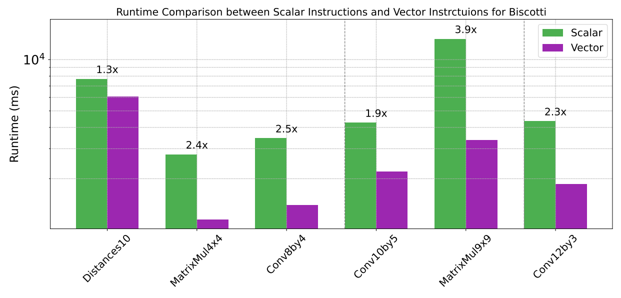


Figure 7: Runtime for Scalar and Vector Instruction of Biscotti

condenses the instruction count to 45 by using a tiling factor of 3. Biscotti coalesces instructions in parallel from highly efficient subprogram schedules which, in general, will always have fewer instructions than schedules generated without tiling factors using Coyote. Finally, Figure 7 compares the runtimes against both scalar and vector instructions for Biscotti. Biscotti always performs better in comparison to scalar code generated from Biscotti, and vector code generated from Coyote.

Table 1: Number of Vector Instructions for Coyote and Biscotti

Benchmark	# BISCOTTI	# COYOTE
Distances10	60	214
MatrixMul4x4	26	91
Conv8by4	37	376
Conv10by5	27	45
MatrixMul9x9	45	426
Conv12by3	81	215

6.4 Tiling Factor Comparison using Biscotti

We answer question 3 we using the following benchmarks.

(1) Convolution with a vector of size 16, kernel size of 5 with a tiling factor of 2, 3 and 4.

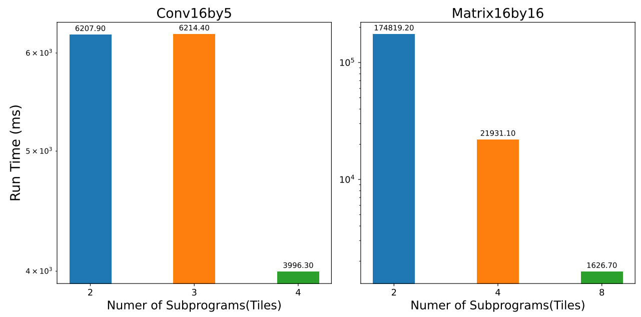


Figure 8: Run Time for Varying Tiling Factors Biscotti

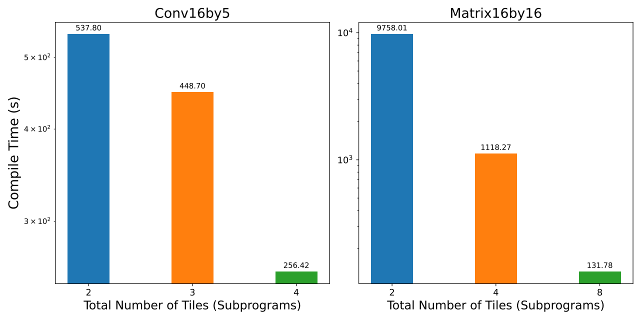


Figure 9: Compile Time for Varying Tiling Factors Biscotti

(2) Matrix multiplication of two 16 × 16 matrices with a tiling factor of 2, 4 and 8.

Figures 8 and 9 suggest that a large tiling factor leads to a small tile size which enhance performance, as they result in simpler subprograms. The reduction in the complexity and size of the subprograms leads to a narrower search space for Coyote. The probability that Coyote is able to converge on a highly efficient schedules is much higher which leads to enhanced compile times and runtimes. However, an interesting observation from Figure 8 from Benchmark 1 (`Conv16by5`) is that the runtime does not improve when increasing the number of subprograms 2 to 3. This could be explained by the minimal reduction in vector instructions (Table 2) down from 79 to 74 when comparing tiling factors 2 and 3. Consequently, the benefit of a smaller search space does not significantly enhance runtime in this instance.

Table 2: Biscotti Vector Instructions with Various Tile Factors

Operation	# BISCOTTI	TILING FACTOR
Conv16by5	79	2
Conv16by5	74	3
Conv16by5	54	4
Matrix16by16	1375	2
Matrix16by16	213	4
Matrix16by16	29	8

While smaller tiles can reduce complexity and potentially speed up the scheduling and compilation process, there is a point of diminishing returns. Over-tiling can lead to excessive decomposition, where the overhead of managing numerous smaller subprograms outweighs the benefits of the improved search space. This balance is crucial as tile sizes that are too small might result in lost vectorization benefits, hence negating performance gains. Conversely, large tile sizes may overwhelm Coyote's ability to efficiently navigate the search space, hindering its capacity to find optimal schedules. The choice of tile size emerges as a critical factor in the efficiency of the vector schedule generated.

7 DISCUSSION

As discussed in Section 2, the *recursion* method can be extended to Biscotti. For a bounded-recursion, we can build a call graph to

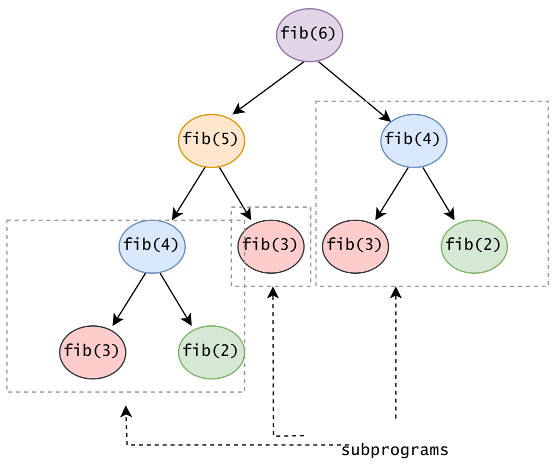
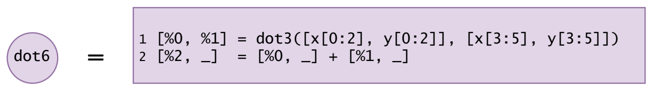


Figure 10: Fibonacci Recursion Call Tree



```
[%0, %1] = dot3([x[0:2], y[0:2]], [x[3:5], y[3:5]])
[%2, _]  = [%0, _] + [%1, _]
```

Figure 11: Schedule for `dot6` composed of `dot3`

a specified depth, and identify sub-graphs to be compiled using Coyote. The sub-graph schedules can then be parallelized, with the necessary reduction steps as defined by the call tree to produce the final result of the program. In Figure 10, Coyote can be used to generate subprograms for the Fibonacci sequences of depth 4 and 3. By using Biscotti these schedules are then interleaved at each depth, to construct the final schedule, and then reduced at each level to integrate the outcomes from the lower depths. We restrict Biscotti's approach to only bounded-depth recursion because all conditionals must be evaluated at compile time to avail any vectorization opportunities (Section 2.2).

The method of Function Decomposition, described in Section 3 can be extended to Biscotti. We can apply this method when we have a function f is composed of functions f_1 and f_2, denoted by $f = f_1 \circ f_2$. Vector schedules for each constituent function, f_1 and f_2, can be generated and compiled independently. Subsequently, a composite schedule that includes calls to both f_1 and f_2 can then be synthesized using Biscotti. For instance, consider the expressions `dot(6) = dot(3) + dot(3)`. Here, the function `dot` is designed to calculate the dot product of a vector. In this context, `dot(6)` refers to the dot product of a 6 element vector. To simplify the computation, this operation is divided into two smaller dot product calculations of 3 element sub-vectors. The final result is obtained by adding the results of these two `dot(3)` operations. Here, schedules for `dot(3)` are generated independently. This strategy is illustrated in Figure 11, which shows the composition for generating a schedule for `dot(6)`.

8 RELATED WORK

There exists a whole host of compilers that target vectorization in FHE [4–7, 9, 12, 13].

Coyote [13], the vectorizing compiler that we use, generates efficient vector schedules by searching a large graph for schedules that minimize rotations.

Porcupine is another vectorizing compiler that uses a sketch-based synthesis approach to generate vectorized kernels given a reference implementation. Porcupine uses heavy-weight synthesis techniques that are designed to work for irregular programs, but it is not automated and requires a programmer-provided sketch as a starting point [4]. The synthesis-based approach makes Porcupine's compile times long (up to many minutes) and limits the size of the programs it can handle. Both Porcupine and Coyote make no assumptions about the programs, but they suffer from poor compilation times. Biscotti targets synthesis-based compilers for FHE and vectorizes *subcircuits* instead of an entire circuit at once, and hence is able to scale better.

CHET is a vectorizing compiler for homomorphic tensor programs that automatically selects encryption parameters and chooses efficient tensor layouts. CHET is designed to optimize dense tensor operations in neural network inference, and does not work well with other types of programs, especially those with irregular computations that are difficult to vectorize [7]. HECO is another compiler that restructures high-level imperative programs to FHE operations through a variety of circuit optimizations to produce vector code [18]. In particular, HECO is optimized for regular programs which include dense loops. While Biscotti can handle such programs (for example, by identifying the loop body as a subcircuit), it is designed for much more general applications.

8.1 General Vectorization Techniques

Loop Vectorization. Loop vectorization is a technique that uses data level parallelism to improve the performance of loops by packing instructions into vectors [11, 16]. Traditional loop vectorizing compilers build a dependence graph with direction and distance information to determine whether a loop is vectorizable. This process involves expanding each operation in the loop from a scalar type to a vector type, which is straightforward for simple loops. However, many computations require more advanced loop transformations to be in a vectorizable form and without dependencies to ensure consistency. Biscotti does not only rely on looping structures and can build schedules for any programs.

Superword Level Parallelism. Superword-Level Parallelism (SLP) is a technique for automatically vectorizing a more general class of programs. SLP iterates over sets of isomorphic scalar instructions and packs them into vectors [10]. SLP works well with irregular programs, because it does not rely on the presence of data-parallel loops. However, SLP does not take into consideration the cost of rotations, nor does it have wide enough vector slots. Modern variants of SLP such as VeGen [3] and goSLP [14] do factor in data movement costs, but cannot fully capture the subtlety of rotation costs in FHE. SuperVectorization, generalizes SLP to uncover parallelism that spans different blocks and loops nests [2]. SuperVectorization analyzes straight line code, and finds vectorization opportunities by subsuming the role of an outer-loop, inner-loop and straight lines vectorizers. Biscotti focuses on finding and scheduling operations to enable parallel execution by determining patterns in a program.

9 CONCLUSION

In this paper, we introduce Biscotti as an optimization that integrates with existing synthesis-based vectorizing compilers for FHE. We evaluate Biscotti on an existing FHE-aware compiler, Coyote. Coyote and other synthesis-based FHE compilers do not scale well with large programs, because they generate large graph spaces of potential schedules to search through, a process that adds significant computational overhead and often does not generate highly efficient schedules. To overcome these limitations, we propose a method for decomposing large programs into smaller, more manageable subprograms, synthesizing schedules for each subprogram, and then consolidating them into a unified vector schedule for the entire program. Our strategy utilizes loop transformation via tiling, which yields substantial improvements in compilation time and runtimes over Coyote. Finally, we also discuss two additional approaches (*recursion* and *function composition*) for program decomposition that can be extended to Biscotti.

REFERENCES

[1] Zvika Brakerski, Craig Gentry, and Shai Halevi. 2013. Packed Ciphertexts in LWE-Based Homomorphic Encryption. (01 2013). https://doi.org/10.1007/978-3-642-36362-7_1

[2] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All you need is superword-level parallelism: systematic control-flow vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3519939.3523701

[3] 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 '21)*. Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[4] 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

[5] 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

[6] 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

[7] 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

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

[9] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481. https://eprint.iacr.org/2020/1481 https://eprint.iacr.org/2020/1481.

[10] 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

[11] S. Larsen, R. Rabbah, and S. Amarasinghe. 2005. Exploiting vector parallelism in software pipelined loops. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. 11 pp.–129. https://doi.org/10.1109/MICRO.2005.20

[12] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. 2022. Hecate: performance-aware scale optimization for homomorphic encryption compiler. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 193–204. https://doi.org/10.1109/CGO53902.2022.9741265

[13] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 118–133. https://doi.org/10.1145/3582016.3582057

[14] 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

[15] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* 48 3 (1970), 443–53. https://api.semanticscholar.org/CorpusID:17406543

[16] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization - revisited for short SIMD architectures. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2–11.

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

[18] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. HECO: Fully Homomorphic Encryption Compiler. arXiv:2202.01649 [cs.CR]

[19] Jingling Xue. 1997. On Tiling as a Loop Transformation. *Parallel Processing Letters* 07, 04 (1997), 409–424. https://doi.org/10.1142/S0129626497000401 arXiv:https://doi.org/10.1142/S0129626497000401