



LLM-Vectorizer: LLM-Based Verified Loop Vectorizer

Jubi Taneja

Microsoft Research
Redmond, USA
jubitanaja@microsoft.com

Avery Laird

University of Toronto
Toronto, Canada
alaird@cs.toronto.edu

Cong Yan

Microsoft Research
Redmond, USA
congyan.me@gmail.com

Madan Musuvathi

Microsoft Research
Redmond, USA
madanm@microsoft.com

Shuvendu K. Lahiri

Microsoft Research
Redmond, USA
shuvendu.lahiri@microsoft.com

Abstract

Vectorization is a powerful optimization technique that significantly boosts the performance of high performance computing applications operating on large data arrays. Despite decades of research on auto-vectorization, compilers frequently miss opportunities to vectorize code. On the other hand, writing vectorized code manually using compiler intrinsics is still a complex, error-prone task that demands deep knowledge of specific architecture and compilers.

In this paper, we evaluate the potential of large-language models (LLMs) to generate vectorized (Single Instruction Multiple Data) code from scalar programs that process individual array elements. We propose a novel finite-state-machine *multi-agents* based approach that harnesses LLMs and test-based feedback to generate vectorized code. Our findings indicate that LLMs are capable of producing high-performance vectorized code with run-time speedup ranging from 1.1x to 9.4x as compared to the state-of-the-art compilers such as Intel Compiler, GCC, and Clang.

To verify the correctness of vectorized code, we use Alive2, a leading bounded translation validation tool for LLVM IR. We describe a few domain-specific techniques to improve the scalability of Alive2 on our benchmark dataset. Overall, our approach is able to verify 38.2% of vectorizations as correct on the TSVC benchmark dataset.

CCS Concepts: • Software and its engineering → Source code generation.

Keywords: Loop Vectorization, Large language model, Translation Validation, AI Agents

ACM Reference Format:

Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K. Lahiri. 2025. LLM-Vectorizer: LLM-Based Verified Loop

Vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25)*, March 01–05, 2025, Las Vegas, NV, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3696443.3708929>

1 Introduction

In today's data-driven world, loop vectorization plays a crucial role in accelerating the high performance computing (HPC) and AI applications. Vectorization allows operations to be performed on entire arrays simultaneously, which is much faster than iterating over elements one-by-one. To this end, many decades of research effort has been invested in automatic vectorization. However, compilers frequently fail to apply such optimizations due to the imprecision of static analysis. Vectorization is driven by data dependence analysis which is difficult to establish precisely due to complexities of code including complex control flow, aliasing, discontinuous memory access [9, 17, 18, 22]. Further, compilers use conservative cost models to determine the profitability of vectorization that often leads to poor choices of optimization [18].

To enable users to exploit vectorization opportunities directly, frameworks such as AVX2 [12, 24] introduce a set of compiler *intrinsics* (C-level functions) that allow invoking the assembly level single instruction multiple data (SIMD) instructions from the C source code. For example, the intrinsic `_mm256_loadu_si256` allows for the loading of 256 bits of integer data from memory into a YMM register, while `_mm256_storeu_si256` enables the storage of 256 bits from a YMM register back to memory. These intrinsics, among others, empower developers to write code that is both performant and maintainable. However, this puts the burden of ensuring the correctness of the vectorized program on the user. Given the subtle semantics of these operations, writing such programs is complex, error-prone and is limited to expert users only.

Recent advances in large-language models (LLMs) have demonstrated the potential to generate and transform code based on natural language instructions [5, 21]. On the other hand, formal verification for LLVM has matured with the advent of tools such as Alive2 [16]. Motivated by these progress,



This work is licensed under a Creative Commons Attribution 4.0 International License.

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1275-3/25/03

<https://doi.org/10.1145/3696443.3708929>

we ask the following question to democratize the creation of vectorized programs by non-experts:

Can advances in LLMs and formal verification be leveraged to automatically optimize scalar C programs into equivalent vectorized programs using AVX2 intrinsics?

In Figure 1, a motivating example is presented from the TSVC benchmark [18]. The input C function, as illustrated in Figure 1(a), represents a scalar and is not vectorized by state-of-the-art compilers, including GCC and Clang. We allow the state-of-the-art compilers to be more aggressive at performing vectorization optimization by adding `restrict` keyword for each pointer in a scalar input program. The `restrict` keyword indicates that pointers do not alias which enabled ICC compiler to vectorize the given input program shown in 1(a) [20]. The prevalent cause for the nonvectorization in compilers, GCC and Clang, is attributed to the compilers' detection of a backward loop-carried dependency that occurs when accessing the array element $a[i+1]$ concurrently with the operation of writing to $a[i]$. This perceived dependency hinders the compilers' ability to apply vectorization to the code segment. However, this dependence is spurious. In the scalar code, the operation of writing to $a[i]$ depends on the read operation of current elements of the arrays a and c at the index i . The operation of writing to $b[i]$ depends on the read operation of the subsequent element in the array a , denoted as $a[i+1]$, current elements of the arrays b and d . The update to $b[i]$ is specifically based on the preceding value of $a[i+1]$, which has not been modified by the first statement in the loop. This indicates the non-existence of cross-dependence that could interfere with the vectorization optimization.

Upon prompting GPT-4 [1] to generate vectorized code targeting AVX2, we obtain the C code utilizing the intrinsics of AVX2, as shown in Figure 1(b). Intuitively, intrinsics such as `_mm256_mullo_epi32` perform element-wise multiplication of 32-bit integers within a 256-bit register, producing lower 32 bits of the result. Likewise, intrinsic `_mm256_add_epi32` perform element-wise addition of eight 32-bit integers within a 256-bit register.

GPT-4 produces vectorized code by loading two different vectors of the array a . The vector a_vec incorporates eight integer elements starting from the index i^{th} and another vector a_next_vec includes eight integer elements starting from the index $(i+1)^{th}$. Subsequently, arithmetic operations are performed on these vectors in parallel, and the results are stored back into the arrays a and b .

We compile the vectorized code in Figure 1(b) with Clang without any additional optimizations to study the effect of the LLM-based vectorized code in isolation. On compilation, LLM-based vector code shows a speedup of 7.35x, and 8.08x as compared to Clang and GCC compilers respectively as shown in Figure 1(c). Although LLMs exhibit *creativity* in

the generation of high-performance vectorized code (in this case), it does not provide any assurance of correctness of the generated code. We introduce LLM-VECTORIZER, an end-to-end tool that leverages LLMs to automatically vectorize code correctly. By ingesting natural language prompts augmented with a scalar C code snippet, LLM-VECTORIZER generates the corresponding vectorized C code. Internally, our design harnesses recent work on *AI-based agents* [25] and orchestrate the interaction between such agents (some of whom may be LLMs in turn) using a finite state machine (FSM). This FSM is designed to reduce the number of LLM invocations and to facilitate the repair of incorrect vectorized code generated by LLM through a feedback loop. Multiple agents rely on simple checksum-based testing as a correctness criterion.

In this paper, we improve the trust on the correctness of the plausible (possibly correct) vectorized code generated by LLM-VECTORIZER, by subjecting it to *bounded translation validation* (a form of symbolic formal refinement checking as implemented in Alive2 [16]). We demonstrate that the verification queries often causes Alive2 to time-out; we provide techniques to scale the verification exploiting domain-specific insights for the case of the vectorization problem.

Overview. We summarize the results of our exploration with LLM-VECTORIZER and Alive2 below.

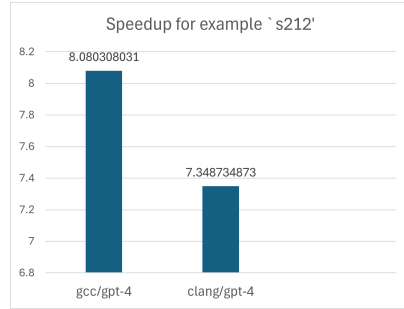
- We find that LLMs are capable of producing efficient vectorized code, often outperforming modern compilers that take a cautious approach to transformation in order to ensure soundness guarantees. (Section 4.3).
- More concretely, LLM-VECTORIZER demonstrates the capability to generate plausibly correct vectorized code for 125 out of 149 test programs in the Test Suite for Vectorizing Compilers (TSVC) benchmark [18], as per the checksum-based testing criterion (Section 2, 4.1.1).
- The use of symbolic verification tool (through Alive2) increases the number of vectorizations that can be proven inequivalent by 2.5x (from 24 to 61) compared to checksum-based testing. Of these, our domain-specific optimizations contribute to proving 20 more vectorizations incorrect.
- Of the 125 plausible cases, we have used Alive2 and our domain-specific optimizations to formally verify (modulo loop unrolling) the correctness for 57 (38.2%) of the test programs. Once again, our optimizations contribute to the verification of 31 additional examples where Alive2 timed out.
- Of the 125 plausible vectorizations, 31 remain inconclusive due to various reasons, including timeouts, out-of-memory issues, unmodeled functions and encoding of vector intrinsics in Alive2. (Section 4.2)
- The multiple agents FSM design in LLM-VECTORIZER identifies 24 new test programs that can be vectorized with just one LLM invocation and also successfully

```

void s212(int n, int *a, int *b, int *c,
         int *d) {
    for (int i = 0; i < n-1; i++) {
        a[i] *= c[i];
        b[i] += a[i + 1] * d[i];
    }
}

```

(a) Unvectorized C code



(c) Runtime Performance Speedup

```

#include <immintrin.h>

void s212(int n, int *a, int *b, int *c, int *d) {
    int i;
    __m256i a_vec, b_vec, c_vec, a_next_vec, d_vec,
    prod_vec, sum_vec;
    for (i = 0; i < n-1 - (n-1)%8; i+=8) {
        a_vec = _mm256_loadu_si256((__m256i *) &a[i]);
        b_vec = _mm256_loadu_si256((__m256i *) &b[i]);
        c_vec = _mm256_loadu_si256((__m256i *) &c[i]);
        a_next_vec = _mm256_loadu_si256((__m256i *) &a[i+1]);
        d_vec = _mm256_loadu_si256((__m256i *) &d[i]);
        prod_vec = _mm256_mullo_epi32(a_vec, c_vec);
        _mm256_storeu_si256((__m256i *) &a[i], prod_vec);
        prod_vec = _mm256_mullo_epi32(a_next_vec, d_vec);
        sum_vec = _mm256_add_epi32(b_vec, prod_vec);
        _mm256_storeu_si256((__m256i *) &b[i], sum_vec);
    }
    for (; i < n-1; i++){
        a[i] *= c[i];
        b[i] += a[i + 1] * d[i];
    }
}

```

(b) Vectorized C Code generated by GPT-4

Figure 1. An example from TSVC benchmark that cannot be vectorized by the state-of-the-art compilers (GCC and Clang) but by ICC and GPT-4.

repairs inequivalent vectorized code using a feedback loop. (Section 4.4)

- We categorize common causes where LLMs encounter challenges in vectorizing programs including loop-carried dependencies and unsafe hoisting. (Section 4.1.3)

Contributions. In summary, the paper makes the following contributions:

1. We evaluate the capabilities of LLMs for the task of performing vectorization through a source-to-source transformation.
2. We describe a novel *AI-agents*-based approach to provide LLMs with feedback to improve the quality of suggestions.
3. We employ formal verification techniques (namely bounded translation validation) to assess the correctness of the transformation at the LLVM level.
4. We provide domain-specific optimizations that help in scaling Alive2 to both verify as well as refute almost twice as many examples.

Organization. The rest of the paper is organized as follows: Section 2 provides the description of LLM-VECTORIZER

including the use of AI-agents. Section 3 describes the symbolic verification using Alive2 as well as our domain-specific optimizations to simplify the task for Alive2. Section 4 discusses the various dimensions for evaluating our approach with different research questions. We discuss related works in Section 5 and conclude in Section 6.

2 Design Overview

LLM-VECTORIZER is an end-to-end tool that uses LLMs to auto-vectorize code correctly. This tool initiates its process by ingesting a prompt provided by the user in natural language, along with a scalar program code snippet written in C. The prompt simply asks for a vectorized program for an AVX2 target for a given input unvectorized program. Upon receiving the input, LLM-VECTORIZER processes the user’s request and proceeds to generate the corresponding vectorized C code. It uses the GPT-4 model to generate code completion.

2.1 Checksum-Based Testing

To ensure the correctness of the vectorized code, our tool, LLM-VECTORIZER uses checksum-based testing to evaluate the equivalence of the original unvectorized and newly vectorized C programs. This is achieved by initializing the input arrays randomly, executing the functions, and comparing

the outputs of both functions. The output produced by LLM-VECTORIZER is proven not equivalent or is marked as *plausible* (possibly correct).

Vectorized codes that appear to be plausible undergo further scrutiny through symbolic verification methods, enhancing the assurance of their correctness. VectorizationGPT’s symbolic verification relies on Alive2, a state-of-the-art translation validation tool for LLVM IR. The high-level workflow of LLM-VECTORIZER is depicted in Figure 2.

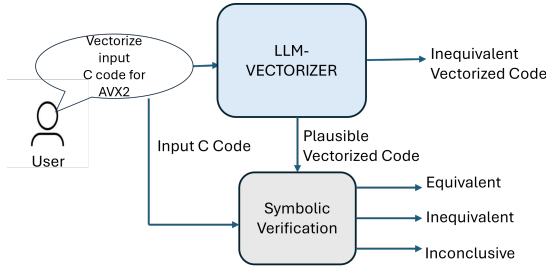


Figure 2. High-level design overview of LLM-VECTORIZER.

2.2 Multi-Agents FSM

In this section, we describe the design of LLM-VECTORIZER that uses *LLM agents* capable of intercommunication and excel in contextual information gathering for problem solving.

2.2.1 Background: Large-Language Model Agents. LLM agents are intelligent entities that act as intermediaries between developers and large language models like GPT-4. These agents are designed to facilitate seamless communication with the underlying language model, enabling developers to harness its power effectively. We can think of them as the bridge that connects human intent with machine-generated text. LLM Agents offer several advantages:

1. LLM agents abstract away the complexities of interacting directly with the raw language model. Instead of dealing with low-level API calls, developers can work with a simplified interface provided by the agent. This abstraction shields them from the intricacies of tokenization, context windows, and other technical details.
2. LLM agents handle context management efficiently. They maintain conversation history, context windows, and user prompts. This context awareness ensures coherent and relevant responses in multiple turns, making interactions more natural.

Recent techniques have harnessed LLM-based agents and compiler feedback for code generation [8].

2.2.2 LLM-VECTORIZER. Our design harnesses these agents and orchestrates them using a finite state machine (FSM), allowing designers to easily specify transitions between agents. This FSM is designed with two primary objectives in mind:

1. reduce the number of LLM invocations required to generate correct vectorized code
2. repair the incorrect vectorized code generated by LLM through a systematic feedback loop

The workflow, illustrated in Figure 3, begins with a user proxy agent initiating a dialogue with a vectorizer assistant agent, providing code for vectorization and dependence analysis information from the Clang compiler, highlighting why Clang cannot vectorize the loop. Dependence analysis identifies data dependence within the loop, i.e. if there is read-after-write, write-after-read data dependence across loop iterations. This analysis also determines loop carried dependence i.e. if an iteration depends on the result of the previous iteration.

The user instructs the vectorizer agent to eliminate the dependence for successful vectorization. Internally, the vectorizer agent consults the LLM and forwards both the original and vectorized code to the compiler tester assistant agent, which employs checksum-based testing to verify the plausibility of the vectorized code. If discrepancies arise, the compiler tester assistant provides feedback to the vectorizer agent, prompting code re-vectorization. This process repeats up to ten times or until a plausible solution is found.

Checksum-based testing involves setting a loop upper bound, initializing input arrays with random values, executing both code versions, and comparing output arrays. If the vectorized code fails to compile, it’s returned to the LLM for correction. Successful code passes through symbolic verification for soundness. In case of inequivalence, the tester agent prompts the vectorizer to resolve discrepancies, ensuring semantic equivalence between vectorized and unvectorized code. The qualitative and quantitative results of the use of the multiagent FSM design are discussed in Section 4.4.

3 Equivalence Checking

Algorithm 1 Algorithm for Formal Equivalence Checking

```

1: procedure CHECKEQUIVALENCE(S, V)
2:   result ← CHECKSUMTESTING(S, V)
3:   if result = INEQUIVALENT then
4:     return result
5:   if result = PLAUSIBLE then
6:     result ← CHECKWITHALIVE2UNROLL(S, V)
7:     if result ≠ INCONCLUSIVE then
8:       return result
9:     result ← CHECKWITHCUNROLL(S, V)
10:    if result ≠ INCONCLUSIVE then
11:      return result
12:    result ← CHECKWITHSPATIALSPLITTING(S, V)
13:  return result

```

In this section, we describe our approach to formally verify the correctness of vectorized code V produced by LLM-VECTORIZER by checking them for equivalence against

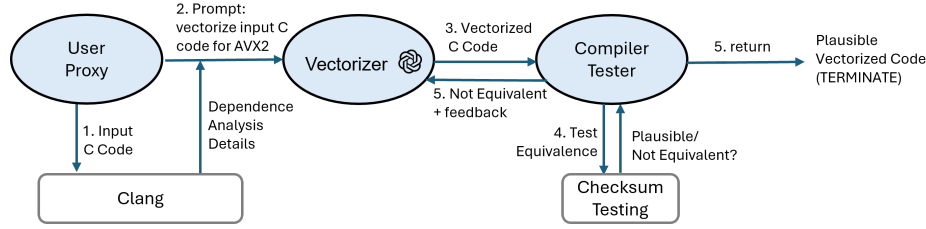


Figure 3. Detailed design of LLM-VECTORIZER.

the input scalar code S . Algorithm 1 describes the overall approach. We assume that the checksum based testing (CHECKSUMTESTING, described in Section 2.1) marks the pair as PLausible (i.e., is unable to distinguish between the two programs using concrete test cases).

3.1 Symbolic Verification with Alive2

For the symbolic verification, we leverage Alive2, an automatic symbolic translator validation tool for the LLVM compiler’s intermediate representation (IR) for checking equivalence. It is designed to ensure that the LLVM compiler’s transformations do not introduce bugs by checking that the optimized target program *refines* the input program. Alive2 provides a comprehensive modeling of *undefined behavior* (UB) in LLVM IR, a critical aspect since LLVM’s optimizers often exploit UB. Alive2 has contributed to clarifying ambiguities in LLVM’s IR specification and has led to numerous bug fixes and patches. Using Alive2 also allows us to avoid defining semantics for vectorization intrinsics, such as `_mm256_blendv_epi8`; Clang is responsible for emitting the proper LLVM IR for such intrinsics, which Alive2 then encodes into the logic of Satisfiability Modulo Theories (SMT-LIB [4]) to be verified by Z3 [7].

Alive2 uses a bounded formal verification, where loops are unrolled a fixed number of times. This has the limitation that bugs that manifest only in a large number of unrolling may be missed by Alive2. Unrolling loops increases the complexity of the verification query sent to the SMT solvers, resulting in the verifier returning INCONCLUSIVE results in the form of timeout or memoryout.

In the next paragraphs, we describe the different challenges encountered to translate the vectorization correctness problem into a bounded translation validation problem (the method CHECKWITHALIVE2UNROLL in Algorithm 1). For this paper, we assume only for loops, where we represent the loops in scalar and vector programs in the canonical form: `for (i = start1; i < end1; i += step1) srcbody`, and `for (i = start2; i < end2; i += step2) vecbody`; it is easy to extend to other ending conditions like `i <= end` or `i != end`, or decrementing iterator such as `i -= step`. We have several examples where `step1` is not 1 and cases where either of both of `end1` and `end2` are complex expressions (see the target program in Fig 1). Alive2 allows unrolling the

loops in the source and target programs before comparing two loop-free programs for equivalence.

Loop alignment. For the vectorized program, a single loop iteration performs updates that are spread over several loop iterations in the scalar program. This implies that the two loop-free programs created for the source and the vectorized program need to perform the same set of updates on both sides. We also fix the unroll factor for the vectorized program to be 1. To compute the unrolling factor for the source program, we compute the least common multiple v of `step1` and `step2`, and unroll source for $m = v/\text{step1}$ times. For simplifying the formal verification, we assume that the vectorized code does not require an epilogue scalar loop as the number of iterations of the scalar program is a multiple of the vectorization width. We ensure this by adding an assumption `(end1 - start1) % m == 0` at the LLVM level through an “assume” instruction, where `%` denotes the modulus operator. For the example in Fig 1, where $m = 8$, we add the constraint `assume (n - 1 - 0) % 8 == 0`. We use a simple static analysis of the C source code and z3 SMT solver to obtain this bound. Our analysis currently does not handle cases where `step1` is not a constant literal, or where the target introduces different number of loops compared to the source program.

Nested loops. For nested loops, we have observed that only the inner loop needs to be vectorized, keeping the outer loop structure similar to the source program. Therefore, for nested loops, we check to make sure that the outer loops are syntactically identical and only align the inner loops. Given the identical outer loops, we only check the equivalence of the inner loops for an arbitrary iteration of the outer loop.

Establishing non-aliasing. To establish the correctness of vectorization one may need to assume some precondition about non-aliasing of the parameters. For example, consider the pair of source and vectorized programs, where the source program is `for (i = 0; i < n; i++) a[i] = b[i] + 1`; and the target program is:

```
for (i = 0; i < n; i += 8) {
    _mm256i b_vec = _mm256_loadu_si256((__m256i *) (b + i));
    _mm256i one_vec = _mm256_set1_epi32(1);
    _mm256i result_vec = _mm256_add_epi32(b_vec, one_vec);
    _mm256_storeu_si256((__m256i *) (a + i), result_vec);
}
```

```

void s124(int *a, int *b, int *c, int *d,
         int *e, int n) {
    int j;
    j = -1;
    for (int i = 0; i < n; i++) {
        if (b[i] > 0) {
            j++;
            a[j] = b[i] + d[i] * e[i];
        } else {
            j++;
            a[j] = c[i] + d[i] * e[i];
        }
    }
}

```

(a) Unvectorized C code

```

#include <immintrin.h>

void s124(int *a, int *b, int *c, int *d, int *e, int n) {
    int j = 0;
    __m256i increment = _mm256_set1_epi32(1);
    __m256i zero = _mm256_setzero_si256();
    for (int i = 0; i < n; i+=8) {
        __m256i vbi = _mm256_loadu_si256((__m256i *)&b[i]);
        __m256i vci = _mm256_loadu_si256((__m256i *)&c[i]);
        __m256i vdi = _mm256_loadu_si256((__m256i *)&d[i]);
        __m256i vei = _mm256_loadu_si256((__m256i *)&e[i]);
        __m256i vprod = _mm256_mullo_epi32(vdi, vei);
        __m256i vsum_b = _mm256_add_epi32(vbi, vprod);
        __m256i vsum_c = _mm256_add_epi32(vci, vprod);
        __m256i vmask = _mm256_cmpgt_epi32(vbi, zero);

        __m256i va = _mm256_blendv_epi8(vsum_c, vsum_b, vmask);
        _mm256_storeu_si256((__m256i *)&a[j], va);
        j+=8;
    }
}

```

(b) Vectorized Code for s124.

Figure 4. Motivating example for Symbolic Verification. In vectorized code (b), `__mm256_blendv_epi8` blends bytes from two 256-bit integer vectors based on the most significant bit of each byte in a mask vector.

Since the addresses of the two arrays `a` and `b` are pointers, Alive2 may consider them to be aliased. If we send this pair to Alive2, it finds a counterexample to equivalence, when `b` aliases with the address of `a-1` (i.e., `&b[1]` is the same pointer as `&a[0]`). For such an input, then source program effectively reduces to `a[i] = a[i-1] + 1` which is not equivalent to the vectorized program, and perhaps not even vectorizable.

To tackle this issue, we ensure that the input arrays are allocated in different memory regions so that addresses of two different arrays never alias. To communicate this to Alive2, we define pointer `a` and `b` as external arrays (instead of function parameter); Alive2 models them as arrays allocated on different memory regions.

Example. Figure 4 shows a motivating example for which checksum based testing CHECKSUMTESTING found the vectorized code to be possibly equivalent PLAUSIBLE, but Alive2 based symbolic verification identifies them as INEQUIVALENT. Figure 4(a) shows the scalar code, which conditionally loads from `c` in the `else` condition. All other arrays are read in both branches. However, the vector code in Figure 4(b) loads eight elements from `c` unconditionally. Such a load may trigger undefined behaviour (specifically in the operation `_mm256_add_epi32`), which Alive2 finds symbolically. However, this bug is missed by checksum testing because such undefined behaviors may not exhibit in all runs. Therefore, symbolic verification is an additional strategy we employ to increase confidence that codes produced by LLM-VECTORIZER are correct.

With the techniques described in this section, we manage to show a few more examples INEQUIVALENT using symbolic formal verification, and prove EQUIVALENT under a bounded unrolling of loops. However, the need to unroll loops often makes Alive2 return INCONCLUSIVE due to timeouts. In the next few subsections, we outline a few domain-specific optimizations to scale the symbolic verification.

3.2 C-Level Unrolling

Given that we have restricted the verification to cases where the number of iterations of the loops in the source scalar program to be a multiple of vectorization width, we can further simplify the verification condition. For example, if the target program has a vectorization width of 8 (e.g., Fig 1), then we know that the loop in the scalar program will execute multiples of 8 (including 0). This means that we can safely skip the check for loop termination in the source program `i < end1` when `i` is not a multiple of 8. We achieve this by using an unrolling at the level of scalar program at the C-level by a preprocessing step before generating the LLVM (instead of Alive2 performing unrolling for the LLVM program).

To unroll the source program for v iterations, we replace the `for` loop as follow: First we initialize `i` as `i = start1`, then unroll the loop body for v times where we add `i+=step1` to the end of each loop body. We perform program analysis and slightly modify the loop body to ensure the unrolled program can compile as follows. (1) We replace "break" statement with "return"; (2) We replace each "goto" label with a

unique label to avoid using repeated "goto" labels across different iterations; finally, (3) we remove duplicated variable declarations.

We also extend the unrolling at the C-level to nested loops. For a nested loop, we unroll only the inner loop. We check whether the outer loop remains exactly the same in source and target, then choose one iteration to verify. We choose an arbitrary iteration of the outer loop by elevating the loop iterator to a parameter value (e.g., if the outer loop is for ($i = 0; i < n; i++$), we remove the for statement and elevate loop iterator i to be a function parameter).

We observe this simple transformation allows the symbolic verification to be scalable and terminate with a non INCONCLUSIVE result. We call this method CHECKWITHCUNROLL in Algorithm 1, and invoke it if the default Alive2 algorithm CHECKWITHALIVE2UNROLL does not succeed.

3.3 Spatial Case Splitting

For several examples without any *loop-carried dependencies*, one can decompose the equivalence checking of an entire array into checking for the equivalence of each index of the array separately. This has the potential to simplify the verification complexity for the SMT solver, at the cost of making multiple verification queries. That is, to check that two arrays $a[0:k]$ and $a'[0:k]$ are equal for some fixed k , one can check k equivalence queries each checking the equivalence of $a[j]$ and $a'[j]$ for each $j \in [0, k]$. Moreover, for the j^{th} query, one can only consider the j^{th} index of the other arrays it reads from — abstracting the content of all the arrays except at index j . Further, in the absence of any loop-carried dependency and ensuring that iteration j (in scalar program) updates j^{th} entry of output arrays reading only from the j^{th} entries of other arrays, we only retain a *single loop iteration* of the scalar program that updates a fixed j^{th} index. To achieve this, at entry we copy the content of each output array (say a) into a local array (say $a1$), and only update the j^{th} index of output array with the value at the j^{th} index in the updated local array before exit (i.e., $a[j] = a1[j]$). The target program, similarly performs the vector operations on the local array (say $a1$) and also updates the output arrays at a single index j . Since we only unroll the vectorized loop once, we generate m equivalence queries, one for each value of $j \in [0, m]$, where m is the vectorization width. We term this as CHECKWITHSPATIALSPLITTING (*spatial case splitting*) in Algorithm 1.

To implement this optimization, we implement a (semi-automated at present) technique to conservatively check that the scalar and the vectorized programs do not have an loop-carry dependencies. In other words, each loop can be processed without any dependence on values from other iterations. Our loop-carry dependency at the C-level checks for the following: (1) the source program accesses only the i^{th} element in *every* array at each i^{th} iteration, and the target accesses only vectors starting at the i^{th} element (e.g.,

`_m256_loadu(&a[i]);`); and (2) both programs do not update any scalar value across loop iteration. Our analysis is syntactic and conservative — it fails to qualify the following program that has no loop-carry dependency for the loop-body $a[i] = a[i+1] + 1$. For example, a semantically equivalent program that copies the original content of a into an array tmp ($tmp[0:n-1] = a[1:n]$) and rewrites the body as $a[i] = tmp[i] + 1$ does not have any loop-carry dependencies.

4 Results

We design our experiments to address the following research questions.

- **RQ1:** How well can GPT4 vectorize the code on its own?
- **RQ2:** Can the vectorized code be formally verified for equivalence?
- **RQ3:** Is the vectorized code performant?
- **RQ4:** Do the AI-agents help improve the quality of generated code?

Dataset. We used the Test Suite for Vectorizing Compilers (TSVC) benchmark to assess the vectorization capabilities of LLM-VECTORIZER. This benchmark consists of 149 for loops that operate on arrays of integer data type exclusively [18]. In the dataset, these loops involve control flow, reduction, and data dependence. Although loops may have a fixed lower bound, the upper bound remains unknown. Furthermore, the step count in the loops varies—either as a constant or as a variable—between different tests. In our evaluation, each for loop is treated as an individual test program.

Experimental Setup. The LLM-VECTORIZER presented in this paper uses the GPT-4 model at its core to generate vectorized code. We do not make any fine-tuning adjustments to the model and use it as is. We configure GPT-4 model with a temperature set to 1.0 to enable more diversity and creativity in the responses. The API version is set to 2023-08-01-preview. For performance evaluation of LLM-generated vectorized code against baseline compilers, we use *Clang-19.0.0git* ([git@github.com:llvm/llvm-project.git 699c408c88b3ed02f25464aa868bd48454fbb3f](https://github.com/llvm/llvm-project.git)), *GCC-10.5.0*, and *ICC-2021.10.0* compilers. The details of the compiler flags to compile unvectorized and vectorized programs are listed in Table 1. In each source program given to a baseline compiler, the restrict keyword is added to every pointer to tell the compiler explicitly that pointers do not overlap (do not alias), allowing compilers to be more aggressive at performing vectorization optimizations. For LLM-Vectorized code, we do not add restrict. We run performance experiments on an eight-core *Intel i7-8650U* CPU with 16GiB RAM and AVX2 target.

Table 1. Compiler Optimization Flags and Version Details.

Compiler	Version	Unvectorized	Vectorized
GCC	10.5.0	-O3 -mavx2 -lm -fno-tree-vectorize	-W -O3 -mavx2 -lm -fno-tree-vectorize -verbose=3 -fno-tree-vectorize -fopt-info -vec-optimized
Clang	19.0.0	-O3 -mavx2 -lm -fno-vectorize -fno-slp -vectorize	-O3 -mavx2 -fstrict-aliasing -fvectorize -fslp-vectorize-aggressive
ICC	2021.10.0	-restrict -std=c99 -O3 -ip -no-vec	-Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize -lm
LLM-VECTORIZER	1.0	-O3 -mavx2 -lm -fno-vectorize -fno-slp -vectorize	-restrict -std=c99 -O3 -ip -vec -xAVX2
			N/A

Table 2. Evaluation of vectorized code using Checksum-based testing.

Parameters	k=1	k=10	k=100
Plausible	72	107	125
Not equivalent	62	40	24
Cannot compile	15	2	0

4.1 [RQ1] Understanding LLM’s Capability to Generate Vectorized Code

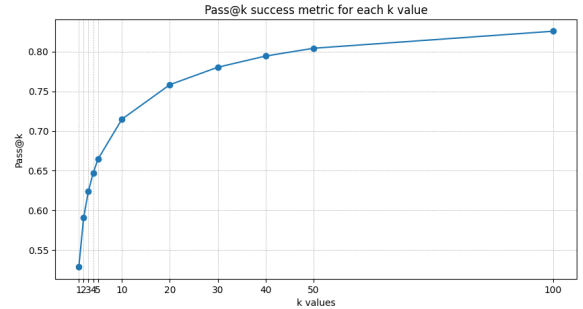
4.1.1 Results from Checksum-based Testing. We conducted an experiment to evaluate the correctness of vectorized C functions generated by LLM. The results of this experiment are summarized in Table 2. We ran this experiment for 149 tests from the TSVC benchmark for varying number of code completions. The column labeled $k = 1$ represent one code completion. In other words, we prompt LLM to generate one output (vectorized program) for each input (scalar) program. We find that 72 tests are plausible. To explore further, we increase the number of code completions to 10 and we find that 107 test programs contain at least one plausible solution. To clarify, we consider a test equivalent if we identify at least one pair of unvectorized and vectorized functions with matching checksum results. However, if all ten vectorized functions generated by GPT-4 fail to be equivalent to the unvectorized function using Checksum-based testing (including cases where any compilation errors occur), we classify the test as not equivalent. If all the candidate solutions for an example fail the compiler, then we mark them under “Cannot compile” row. The $k = 10$ column in Table 2 indicates that 107 tests contain at least one plausible solution. In order to reduce the number of inequivalent tests in a given dataset of 149 test programs, we continue to increase the number of code completions to 100. We found that 125 tests have at least one plausible vectorized program as per the checksum-based testing criterion.

4.1.2 Measuring Success Metric, $\text{pass}@k$. To evaluate the quality of an LLM for the task of code generation (from natural language), while accounting for the statistical non-deterministic nature of LLMs, Chen et al. [5] introduced the $\text{pass}@k$ metric. Assuming the presence of a set of validation tests T , a code suggestion is considered “correct” if it satisfies all the tests in T . Given a sufficiently large number of sample

size n , they defined $\text{pass}@k$ as the expected mean for a sample of size k (k usually smaller than n) to contain at least one correct solution. In our case, we adapt the metric to the case when a code suggestion is labeled PLAUSIBLE by the Checksum-based testing.

For each test program in the TSVC benchmark, we compute the ‘ $\text{pass}@k$ ’ value for different values of k (for example, 1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100) as shown on the x-axis of the chart in Figure 5. At the end, we compute the average ‘ $\text{pass}@k$ ’ value over 149 tests in the dataset for each ‘ k ’.

In Figure 5, the $\text{pass}@k$ metric shows a steep increase as k values rise from 1 to 20, indicating a rapid improvement in the LLM-VECTORIZER’s performance with a small number of attempts. The growth in $\text{pass}@k$ begins to slow beyond $k = 20$, suggesting diminishing returns on the LLM-VECTORIZER’s ability to generate additional correct vectorized code. As k approaches 50, the $\text{pass}@k$ metric reaches near-maximum, indicating that increasing k beyond this point yields minimal improvement in correct vectorized code generation.

**Figure 5.** Effectiveness of LLM-VECTORIZER at generating correct vectorized code at different levels of k .

4.1.3 Qualitative Analysis of LLM’s Shortcomings. The specific reasons for different slowdowns are diverse; however, at a high level, slowdowns occur because a compiler produces faster code than LLM-VECTORIZER. This happens in two main ways: (1) vectorization is not profitable, or (2) compilers produce higher quality vector code than LLM-VECTORIZER. Crucially, we force LLM-VECTORIZER to always produce vector code, even though vectorization is often not the best choice. GCC, Clang, and ICC all have cost models

to estimate whether vectorization is profitable. For example, `s482` experiences one of the largest slowdowns, mainly because vectorization was not profitable; even unvectorized compiler code is faster than vectorized compiler code. In other cases, such as reductions as noted in the paper, compilers are highly optimized for particular code patterns, and it is hard to produce faster code.

We further present a discussion about which codes LLM-VECTORIZER could not be vectorized. In total, LLM-VECTORIZER failed to vectorize 24 tests. Dependence is the main shared attribute among failed tests; for example, many tests contained loop-carried dependences. By providing a simple dependence analysis through Clang feedback (step 1 labeled in Figure 3) and input output examples through checksum-based testing (step 5 labeled in Figure 3), LLM-VECTORIZER was able to succeed in more cases, as shown in Section 4.4. Despite this, LLM-VECTORIZER still struggled with dependence relationships. In the following, we explain the two most common difficulties we have observed.

One-time Dependence. Some loops contained a possible loop-carried dependence that could be satisfied at most once during execution. A common solution to these cases is loop *splitting*, or *distribution*, which partitions the loop domain in order to break up the dependence. We could not consistently get LLM-VECTORIZER to perform the required transformations to vectorize such codes.

Unsafe Hoisting. Hoisting instructions out of a loop body is not always a safe transformation. In several tests, hoisting instructions resulted in easier to vectorize loops, but altered the code semantics. Due to poor dependence analysis, LLM-VECTORIZER performed several unsafe hoists that could not be successfully repaired.

4.2 [RQ2] Can LLM-Generated Vectorized Code be Formally Verified?

In this section, we assess the formal verifiability of the 125 tests proven plausible using checksum-based testing. Initially, these 125 tests are categorized as inconclusive, pending formal verification of their correctness.

Table 3 presents a summary of the results from various equivalence checking methodologies. The symbolic verification employing the out-of-the-box Alive2 unroll technique, described as `CHECKWITHALIVE2UNROLL` in Section 3.1, proves the correctness of vectorization for 26 tests, leaving 82 tests inconclusive due to multiple factors such as timeouts, out of memory, and unrecognized AVX2 intrinsics not yet encoded in Alive2.

Subsequently, we leverage the unrolling at the C program level for the unvectorized program (`CHECKWITHCUNROLL` in Section 3.2) within the subset of 82 test programs that remained unproven by the preceding technique. This optimization enabled us to conclusively verify 18 as non-equivalent, and 28 tests as equivalent (modulo loop unrolling). Notably,

Table 3. Evaluation of Vectorized code using equivalence checking techniques. "Total" is total number of test programs, "Equiv", "Not Equiv" and "Inconcl" refer to EQUIVALENT, IN-EQUIVALENT and INCONCLUSIVE respectively.

Techniques	Total	Equiv	Not Equiv	Inconcl
Checksum	149	0	24	125
Alive2	125	26	17	82
C-Unroll	82	28	18	36
Splitting	36	3	2	31
All	149	57	61	31

this method significantly reduced the incidence of timeouts by simplifying the queries sent to the SMT solver.

Finally, we applied the spatial-case splitting technique (`CHECKWITHSPATIALSPLITTING` in Section 3.3) to the remaining dataset of 36 tests. Unfortunately, a majority of the programs were filtered away due to our conservative static analysis to ensure lack of loop-carried dependencies. After this selective filtering, we were left with five test programs, of which three were validated as equivalent and two as non-equivalent.

In conclusion, we can formally verify 57 test programs as correct, refute 61 tests, leaving 31 as inconclusive. Of these, our domain-specific optimizations contributed to helping Alive2 refute 20 more vectorizations incorrect, and verify 31 additional examples.

4.3 [RQ3] Is LLM-Vectorized Code Faster?

In this section, we compute the runtime performance speedup for 57 equivalent tests proven by equivalence checking. Our focus is on comparing the vectorization capability of mainstream compilers against LLM-VECTORIZER. We found that LLMs are capable of producing high-performance vectorized code with run-time speedup ranging from 1.1x to 9.4x as compared to the state-of-the-art compilers. We classify each test into one of six categories, each represented by a different color in Figure 6. Each test is associated with three different bars to show the performance of LLM-VECTORIZER against GCC, Clang, and ICC, respectively. The baseline speedup is set at 1.0, representing the scenario where state-of-the-art compiler and LLM-VECTORIZER exhibit equivalent performance. Any speedup value below 1.0 indicates a slowdown in LLM-VECTORIZER's performance relative to the other compiler. The following paragraphs provide an analysis of each category.

Control Flow. LLM-VECTORIZER shows the highest performance benefits in test `s278` (for loop shown below), which contains complicated control flow with `goto` statements that requires selection instructions to vectorize efficiently. Otherwise, general-purpose compilers can vectorize many codes with control flow by using `if-conversion`; therefore, other

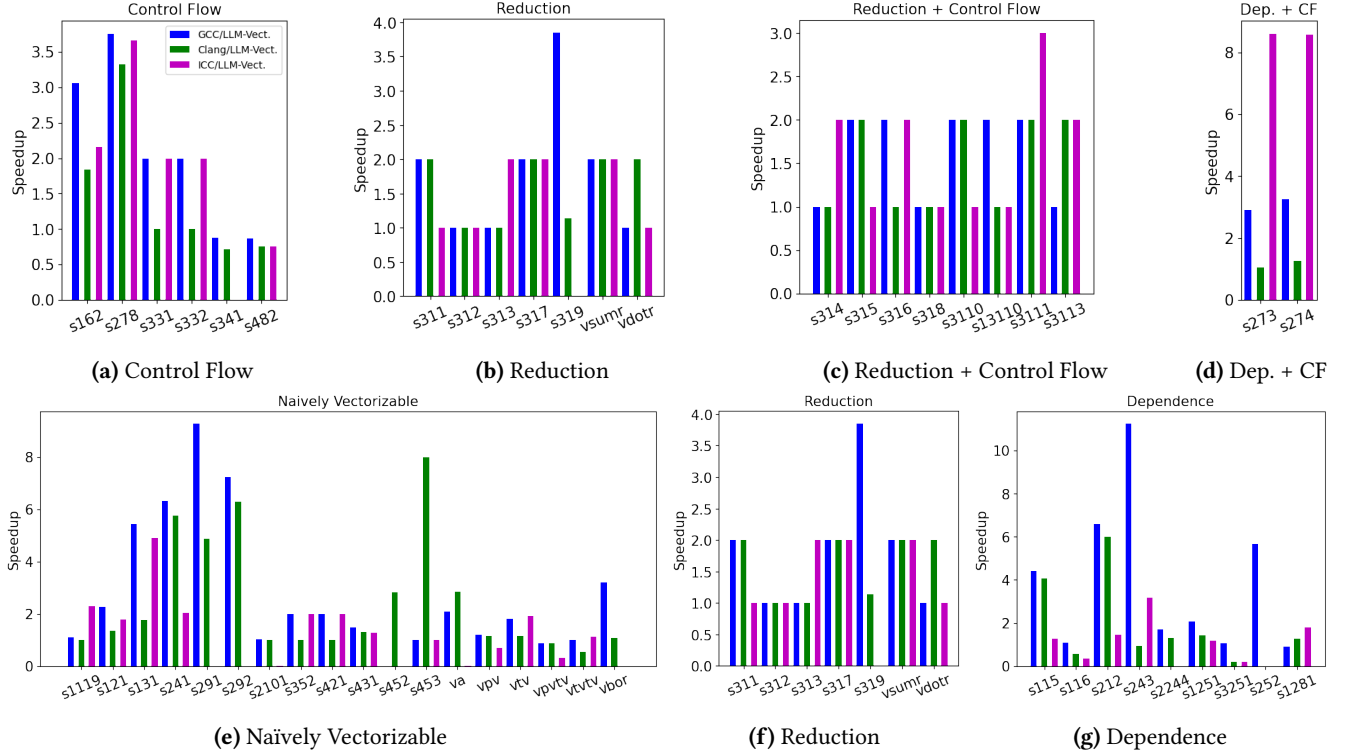


Figure 6. Runtime performance speedup for formally correct test programs from TSVC benchmark.

tests in this category show no speedup, or some slowdown, where the compiler does a better job of vectorization.

```
for (int i = 0; i < n; i++) {
    if (a[i] > 0) {
        goto L20;
    }
    b[i] = -b[i] + d[i] * e[i];
    goto L30;
L20:
    c[i] = -c[i] + d[i] * e[i];
L30:
    a[i] = b[i] + c[i] * d[i];
}
```

Dependence. Unlike control flow, which is usually straightforwardly vectorized by compilers, loops with dependence can disable vectorization entirely due to imprecise dependence analysis. LLM-VECTORIZER therefore shows several significant speedups in this category. In most cases, LLM-VECTORIZER has the largest speedups over GCC and Clang, and the smallest speedups (or slowdowns) over ICC. Because ICC performs a sophisticated dependence analysis that is tightly integrated with the loop vectorizer, it tends to produce fast vector code, even with dependences. However, GCC and Clang often disable vectorization entirely.

Dependence+Control Flow. A mixture of dependences and control flow are difficult for compilers to vectorize; therefore, LLM-VECTORIZER’s more aggressive vectorization strategy always attains speedups in this category. For example, we include s274 below:

```
for (int i = 0; i < n; i++) {
    a[i] = c[i] + e[i] * d[i];
    if (a[i] > 0)
        b[i] = a[i] + b[i];
    else
        a[i] = d[i] * e[i];
}
```

In this case, control flow is mixed with loop-carried dependencies involving a.

Naïvely Vectorizable. This category is a broad grouping of loops that are generally “easy” to vectorize for most compilers; they do not contain any complicated features like control flow or dependence relations. Because compilers produce efficient vector code for such loops, the most number of slowdowns and negligible speedups for LLM-VECTORIZER are observed in this category. Notable exceptions include tests s291 (shown below) and s292, which although do not contain control flow or dependence, require transformations such as loop peeling for the best performance. ICC is able to achieve such performance automatically, while GCC and Clang cannot.

```
int im1;
im1 = n-1;
for (int i = 0; i < n; i++) {
    a[i] = (b[i] + b[im1]) * 2;
```

```
    im1 = i;
}
```

Reduction. Unlike naïvely vectorizable loops, reductions involve loop-carried dependence that requires special handling to vectorize. However, the reduction pattern is so common that compiler support for reductions is robust. This is shown in Figure 6 by either small speedups or slowdowns for LLM-VECTORIZER. For example, below we include for loop from test *vsumr*:

```
sum = 0.;
for (int i = 0; i < n; i++)
    sum += a[i];
```

Such a straight-forward reduction pattern is very well supported by vectorization passes in general-purpose compilers.

Reduction+Control Flow. Reductions mixed with control flow can be vectorized by applying if-conversion. Therefore, this category is also amenable to traditional compiler vectorization techniques, and LLM-VECTORIZER’s speedups are less significant. However, up to a 2× speedup is still obtained over ICC.

4.4 [RQ4] Evaluation of Multi-Agent FSM

While addressing the initial research question delineated in Section 4.1.1, our findings indicate that 125 tests have been validated as plausible. However, this required up to 100 LLM invocations to ascertain an equivalent vectorized code for numerous test programs, which proved to be very expensive. In this section, we aim to assess whether the implementation of a multiple agents finite state machine, in conjunction with auxiliary tools, can effectively reduce the frequency of LLM invocations. Additionally, we will explore the potential of multiple agents in facilitating the repair of non-equivalent vectorized programs through a feedback loop mechanism.

4.4.1 Number of LLM Invocations. In our study, using the checksum-based testing criterion of equivalence, we identified 72 plausible tests when generating only code completions, as detailed in Table 2. Notably, we discovered 24 new tests that were previously unsuccessful with a single code completion but are now successfully generated using multiple agents FSM. Consequently, the total number of plausible tests increased to 96 when considering just one LLM invocation, compared to the initial 72 tests. Furthermore, despite generating 100 code completions, we encountered four test programs that did not have a plausible vectorization previously but now have a plausible vectorization with just a single code completion in multi-agents design. These findings suggest that the design involving multiple agents along with auxiliary tools holds promise for reducing the number of LLM calls and improving the quality of the generation.

4.4.2 Qualitative Analysis of Multiple Agents towards Repair. In this section, we study the capability of multiple agents by taking one example in facilitating the repair of non-equivalent vectorized programs through a feedback loop mechanism. For the function *s453*, the multi-agents FSM

takes two attempts to successfully generate vectorized code. The input scalar program is shown here.

```
void s453(int *a, int *b, int n) {
    int s = 0;
    for (int i = 0; i < n; i++) {
        s += 2;
        a[i] = s * b[i];
    }
}
```

First Attempt. The vectorizer assistant agent starts by naïvely transforming scalar operation $s+ = 2$ to a SIMD operation. It initializes a vector *s_vec* to zero, which is used to store the scalar *s* in vector form. It sets up a vector *two_vec* with all elements as 2, which is used to increment *s_vec* in each iteration. Subsequently, it vectorizes all other operations and increments *s* by 16, corresponding to the simultaneous processing of eight elements. Here is the vectorized code:

```
void s453(int *a, int *b, int n) {
    __m256i s_vec = _mm256_set1_epi32(0);
    __m256i two_vec = _mm256_set1_epi32(2);
    __m256i s_increment = _mm256_set1_epi32(16);
    int i = 0;
    for (; i <= n - 8; i += 8) {
        s_vec = _mm256_add_epi32(s_vec, two_vec);
        __m256i b_vec = _mm256_loadu_si256((__m256i*)&b[i]);
        __m256i a_vec = _mm256_mullo_epi32(s_vec, b_vec);
        _mm256_storeu_si256((__m256i*)&a[i], a_vec);
        s_vec = _mm256_add_epi32(s_vec, s_increment);
    }
}
```

When compiler tester agent tests the equivalence using checksum-based testing, it finds a discrepancy in the actual and expected output arrays. It sends the feedback to the vectorizer agent. The vectorized code in the first attempt is not equivalent to the original scalar code because it updates value of *s* to 2 in a vector slice comprising of eight elements. In contrast, the scalar code updates *s* by 2 in every iteration, which means the values of *s* should be updated to 2, 4, 6, 8, and so on. Let’s see how vectorizer understands this issue and fixes it in the second attempt.

Second Attempt. The vectorizer agent uses the input and output values of arrays and attempts to vectorize again. This time, it vectorizes correctly by initializing *s_vec* with a set of values as expected. There is no need to increment each value in *s_vec* by two anymore, but just by 16 to operate on next slice of eight elements. Here is the vectorized code:

```
void s453(int *a, int *b, int n) {
    __m256i s_vec = _mm256_setr_epi32(2, 4, 6, 8, 10, 12, 14, 16);
    __m256i two_vec = _mm256_set1_epi32(16);
    int i = 0;
    for (; i <= n - 8; i += 8) {
        __m256i b_vec = _mm256_loadu_si256((__m256i*)&b[i]);
        __m256i a_vec = _mm256_mullo_epi32(s_vec, b_vec);
        _mm256_storeu_si256((__m256i*)&a[i], a_vec);
        s_vec = _mm256_add_epi32(s_vec, two_vec);
    }
}
```

The compiler tester agent finds this solution to be correct. The process terminates and the tester returns the correct

vectorized code. This example is also formally verified and proven equivalent using Alive2 unroll technique.

We ran this experiment on 149 tests in the TSVC benchmark and configured multi-agent FSM to allow a maximum of ten attempts to solve the problem. We found correct vectorized codes for the 92 tests. In the evaluation of 92 tests, agents required multiple iterations to resolve nine of them. The maximum number of attempts recorded to successfully address an issue within a single test was seven.

5 Related Work

Compiler auto-vectorization. Maleki et al. [18] studies the efficacy of compilers in vectorizing a synthetic benchmark and real applications, revealing that a significant portion of loops remain unvectorized, highlighting the complexity of auto-vectorization in the presence of intricate control flows. Siso et al. [22] assess compilers' vectorization capabilities by systematically withholding information that aids the auto-vectorization process, thereby providing a more realistic gauge of compilers' performance in practical scenarios. Allen et al. [2] propose a systematic method to convert control dependence into data dependence, facilitates the application of data dependence-based program transformations.

ML-guided Compiler Optimizations. In recent years, the use of data-driven approaches to compiler optimizations has been explored as an alternative to traditional rule-based heuristics. NeuroVectorizer[11] utilizes an end-to-end deep reinforcement learning based approach for vectorization decisions. Mendis et al. [19] leverages imitation learning to replicate the decisions of an optimal Integer Linear Programming (ILP) model, generating vector code that is both efficient and functionally equivalent. Ashouri et al. [3] integrates a machine learning model with LLVM's inlining process, and predicts the potential performance gains from inlining decisions. More recently, Cummins et al. [6] leverage fine-tuned LLMs to enhance compiler heuristics, aiming to optimize program execution without the need for extensive manual tuning. Grubisic et al. [10] extend this line of work by exploring the use of real-time compiler feedback for LLMs to learn and improve subsequent compilation.

In contrast to all the prior uses of ML/LLM for compiler optimization, we use the emergent capabilities of LLMs such as GPT-4 to directly perform a source-to-source rewriting of entire loop bodies. Unlike traditional compiler optimizations, these transformations can be incorrect. Hence we propose employing testing and formal verification techniques to verify the correctness of these transformations.

Formal verification for compilers. In addition to Alive2, translation validation is implemented by a diverse range of existing tools [13, 15, 16, 23]. Although some of these techniques offer the potential to perform unbounded verification (e.g., through equality saturation in Stepp et al. [23]), they are

not as automated and robust as Alive2 for the entire LLVM instruction set.

6 Conclusion

In this paper, we developed LLM-VECTORIZER to evaluate the capabilities of LLMs and AI-based agents towards generating correct and efficient loop vectorization optimizations leveraging compiler intrinsics. Our study highlights opportunities and challenges to automate vectorizing programs through LLMs and formal verification. For future work, we plan to add feedback of failure and rootcause [14] from the equivalence failure within LLM-VECTORIZER, as well as extend the assurance of vectorized code by automating unbounded translation validation.

Acknowledgments

The authors extend their sincere appreciation to Saeed Maleki for providing guidance and insight on vectorization. The authors also thank Sarah Fakhoury for her advice on the design of AI agents and Alive2 developers including Nuno Lopes and Zhengyang Liu for their clarifications and detailed review on the semantics of LLVM IR.

References

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altmenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
- [2] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 177–189. <https://doi.org/10.1145/567067.5670>
- [3] Amir H Ashouri, Mostafa Elhoushi, Yuzhe Hua, Xiang Wang, Muhammad Asif Manzoor, Bryan Chan, and Yaoqing Gao. 2022. Mlgop-erf: An ML guided inliner to optimize performance. *arXiv preprint arXiv:2207.08389* (2022).
- [4] Clark W. Barrett, Leonardo Mendonça de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Initiative and the Rise of SMT - (HVC 2010 Award Talk). In *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010 (Lecture Notes in Computer Science, Vol. 6504)*, Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz (Eds.). Springer, 3.
- [5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [6] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062* (2023).
- [7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
- [8] Sarah Fakhoury, Markus Kuppe, Shuvendu K. Lahiri, Tahina Ramanamandro, and Nikhil Swamy. 2024. 3DGen: AI-Assisted Generation of Provably Correct Binary Format Parsers. *arXiv:2404.10362* [cs.SE]
- [9] Jing Ge Feng, Ye Ping He, and Qiu Ming Tao. 2021. Evaluation of compilers' capability of automatic vectorization based on source code

- analysis. *Scientific Programming* 2021 (2021), 1–15. <https://doi.org/10.1155/2021/3264624>
- [10] Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. 2024. Compiler generated feedback for Large Language Models. *arXiv preprint arXiv:2403.14714* (2024).
- [11] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255. <https://doi.org/10.1145/3368826.3377928>
- [12] Intel. 2024. Intel Advanced Vector Extensions 2. <https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/intrinsics-for-avx2.html>.
- [13] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S Adve, and Grigore Roşu. 2021. Language-parametric compiler validation with application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1004–1019. <https://doi.org/10.1145/3445814.3446751>
- [14] Shuvendu K. Lahiri, Rohit Sinha, and Chris Hawblitzel. 2015. Automatic Rootcausing for Program Equivalence Failures in Binaries. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 362–379. https://doi.org/10.1007/978-3-319-21690-4_21
- [15] Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning* 41 (2008), 1–31. <https://doi.org/10.1007/s10817-008-9099-0>
- [16] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79. <https://doi.org/10.1145/3453483.3454030>
- [17] Mamta Madan, Pradyumn Nand, and Joy Aneja. 2021. Auto Vectorisation Capabilities of the Compilers. In *Proceedings of 3rd International Conference on Computing Informatics and Networks: ICCIN 2020*. Springer, 647–655.
- [18] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. 2011. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 372–382. <https://doi.org/10.1109/PACT.2011.68>
- [19] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. 2019. Compiler auto-vectorization with imitation learning. *Advances in Neural Information Processing Systems* 32 (2019).
- [20] Rodrigo CO Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís FW Góes, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Vectorization-aware loop unrolling with seed forwarding. In *Proceedings of the 29th International Conference on Compiler Construction*. 1–13. <https://doi.org/10.1145/3377555.3377890>
- [21] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020).
- [22] Sergi Siso, Wes Armour, and Jeyarajan Thiyyagalingam. 2019. Evaluating auto-vectorizing compilers through objective withdrawal of useful information. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–23.
- [23] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 737–742.
- [24] Wikipedia. 2024. Advanced Vector Extensions. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions.
- [25] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155* (2023).

Received 2024-05-30; accepted 2024-07-22