LLM-VECTORIZER: LLM-based Verified Loop vectorizer

Jubi Taneja

Research in Software Engineering, Microsoft Research

Collaborators: Avery Laird, Cong Yan, Madan Musuvathi, Shuvendu Lahiri

Email: jubitaneja@microsoft.com
Twitter, GitHub, LinkedIn: @jubitaneja
Mastodon: @jubitaneja@discuss.systems

Programmer’s Dilemma in the World of Compute

Problem: Write fast code.

Solution:
1. Write unoptimized code and leave the rest on compiler optimizations (such as Auto Vectorization)

2. Write SIMD code by hand

Challenges

Problem in Auto-Vectorizers

- Conservative approach
- Misses a lot of optimization opportunities
- Error-prone

Problem in manually writing SIMD code

- Time-consuming
- Error-prone
- Limited to expert programmers who understand vector intrinsics

Problem Statement

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

Motivating Example

```
void src36(int n, int * restrict a, int * restrict b,
           int * restrict c, int * restrict d) {
     for (int i = 0; i < n-1; i++) {
          a[i] *= c[i];
          b[i] += a[i+1] * d[i];
     }
}
```

Vectorize with Clang

```
argument]
../test/final-results/perf/src.c:14:17: remark: loop not vectorized: unsafe dependent memory operations in loop. Use
#pragma clang loop distribute(enable) to allow loop distribution to attempt to isolate the offending operations into
a separate loop
Backward loop carried data dependence. Memory location is the same as accessed at ../test/final-results/perf/src.c:13
:9 [-Rpass-analysis=loop-vectorize]
   14 |         b[i] += a[i + 1] * d[i];
      |                  ^
../test/final-results/perf/src.c:12:5: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
   12 |     for (int i = 0; i < n-1; i++) {
      |     ^
../test/final-results/perf/src.c:13:14: remark: loop not vectorized: value that could not be identified as reduction
is used outside the loop [-Rpass-analysis=loop-vectorize]
   13 |         a[i] *= c[i];
      |             ^
```

$ clang -O3 -mavx2 -fstrict-aliasing -fvectorize -fslp-vectorize-aggressive -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize -lm test.c

Vectorize with GCC

```
jubi@DESKTOP-A49C4F9:~/rise/vectorizationGPT/src$ gcc -W -O3 -mavx2 -lm -ftree-vectorizer-verbose=3 -ftree-vectorize
-fopt-info-vec-missed ../test/final-results/perf/src.c -o src36
../test/final-results/perf/src.c:12:5: missed: couldn't vectorize loop
../test/final-results/perf/src.c:13:14: missed: not vectorized, possible dependence between data-refs *_3 and *_12
../test/final-results/perf/src.c:52:5: missed: couldn't vectorize loop
../test/final-results/perf/src.c:13:14: missed: not vectorized: complicated access pattern.
../test/final-results/perf/src.c:12:5: missed: couldn't vectorize loop
../test/final-results/perf/src.c:13:14: missed: not vectorized, possible dependence between data-refs *_18 and *_27
../test/final-results/perf/src.c:51:13: missed: statement clobbers memory: start_8 = clock ();
../test/final-results/perf/src.c:55:11: missed: statement clobbers memory: end_10 = clock ();
/usr/include/x86_64-linux-gnu/bits/stdio2.h:100:10: missed: statement clobbers memory: __fprintf_chk (stdout.0_3, 1,
"%f", cpu_time_used_11);
```

$ gcc -W -O3 -mavx2 -lm -ftree-vectorizer-verbose=3 -ftree-vectorize -fopt-info-vec-optimized test.c

Vectorize with GPT-4

```c
#include <immintrin.h>
void src36(int n, int *a, int *b, int *c, int *d) {
    int i = 0;
    for (; i <= n - 2 - 7; i += 8) {
        __m256i va = _mm256_loadu_si256((__m256i *)&a[i]);
        __m256i va_next = _mm256_loadu_si256((__m256i *)&a[i + 1]);
        __m256i vc = _mm256_loadu_si256((__m256i *)&c[i]);
        __m256i vd = _mm256_loadu_si256((__m256i *)&d[i]);
        __m256i vb = _mm256_loadu_si256((__m256i *)&b[i]);

        va = _mm256_mullo_epi32(va, vc);
        _mm256_storeu_si256((__m256i *)&a[i], va);

        va_next = _mm256_mullo_epi32(va_next, vd);
        vb = _mm256_add_epi32(vb, va_next);
        _mm256_storeu_si256((__m256i *)&b[i], vb);
    }
    for (; i < n - 1; i++) {
        a[i] *= c[i];
        b[i] += a[i + 1] * d[i];
    }
}
```

Is there a backward loop carried dependence?

```
void src36(int n, int * restrict a, int * restrict b,
           int * restrict c, int * restrict d) {
    for (int i = 0; i < n-1; i++) {
        a[i] *= c[i];
        b[i] += a[i+1] * d[i];
    }
}
```

Is the LLM-based vectorized code correct?

Checksum-based Testing

- Randomly initialize the input arrays
- Execute the function
- Compare the output values written into the arrays from both unvectorized and vectorized programs.

Is the testing criterion sufficient?

Motivating Example for Symbolic Verification

```
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];
        }
    }
}
```

```c
#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;
    }
}
```

Testing ✓

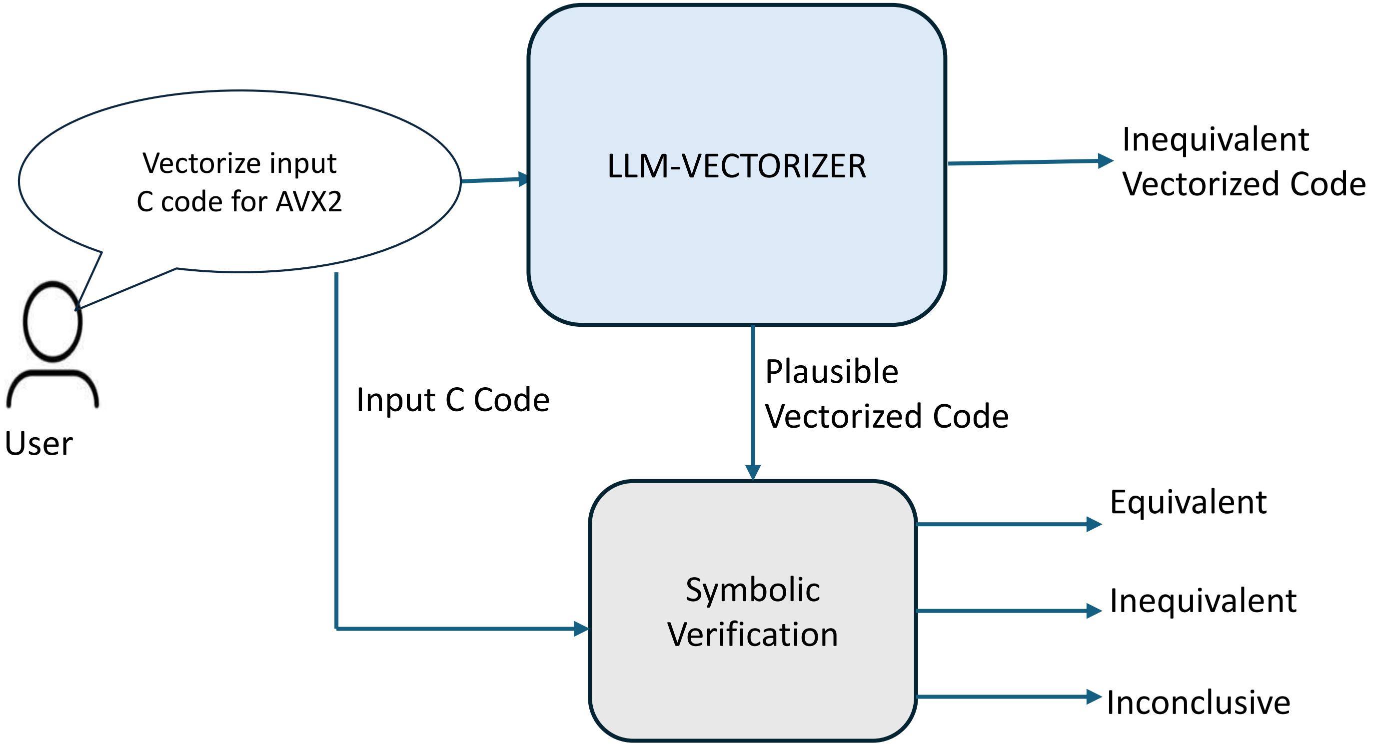
Symbolic Verification ✗

```c
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];
        }
    }
}
```

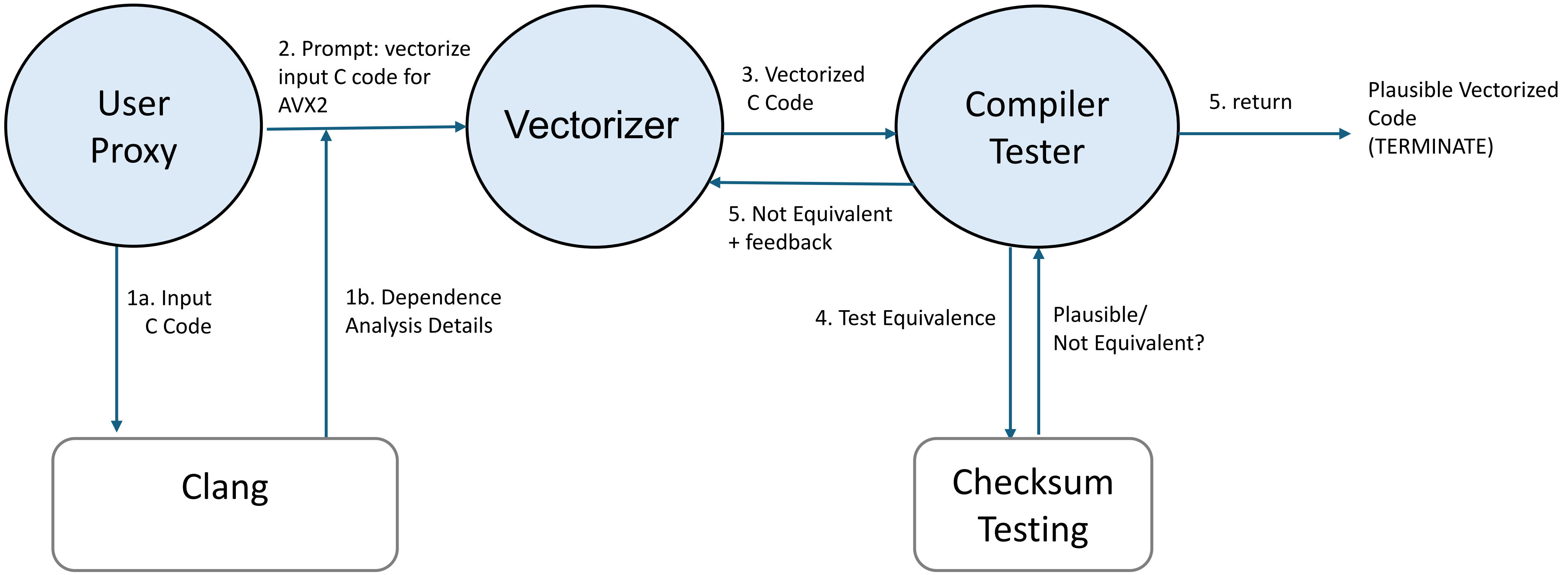
```c
#include <immintrin.h>

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;
    }
}
```

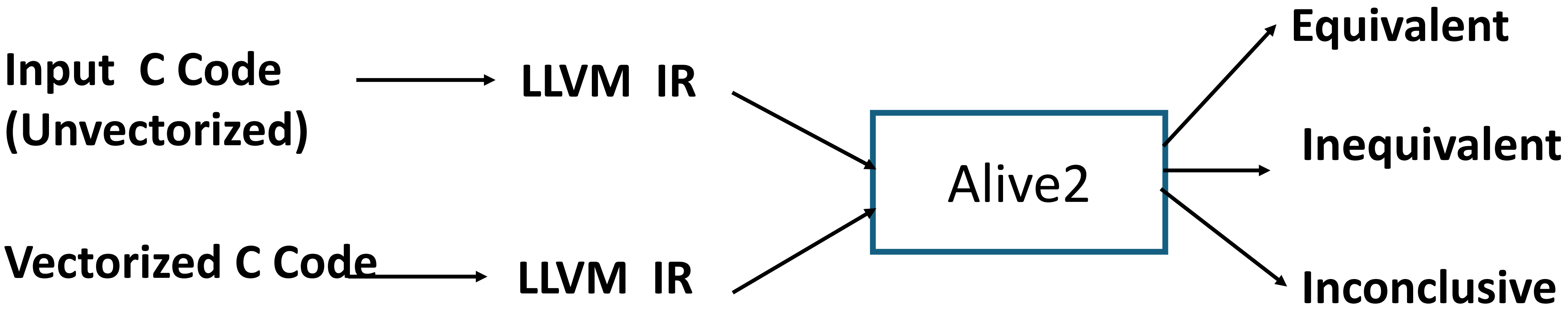
Overview of LLM-VECTORIZER



Detailed Overview of LLM-VECTORIZER



Symbolic Verification with Alive2



Symbolic Verification with Alive2

```
for (int i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
}
```

```
for (int i = 0; i < n - 8; i += 8) {
b_vec = _mm256_loadu_si256 (( __m256i *) &b[i]);
c_vec = _mm256_loadu_si256 (( __m256i *) &c[i]);
a_vec = _mm256_add_epi32 (b_vec , c_vec );
_mm256_storeu_si256 (( __m256i *) &a[i], a_vec );
}
```

Symbolic Verification with Alive2

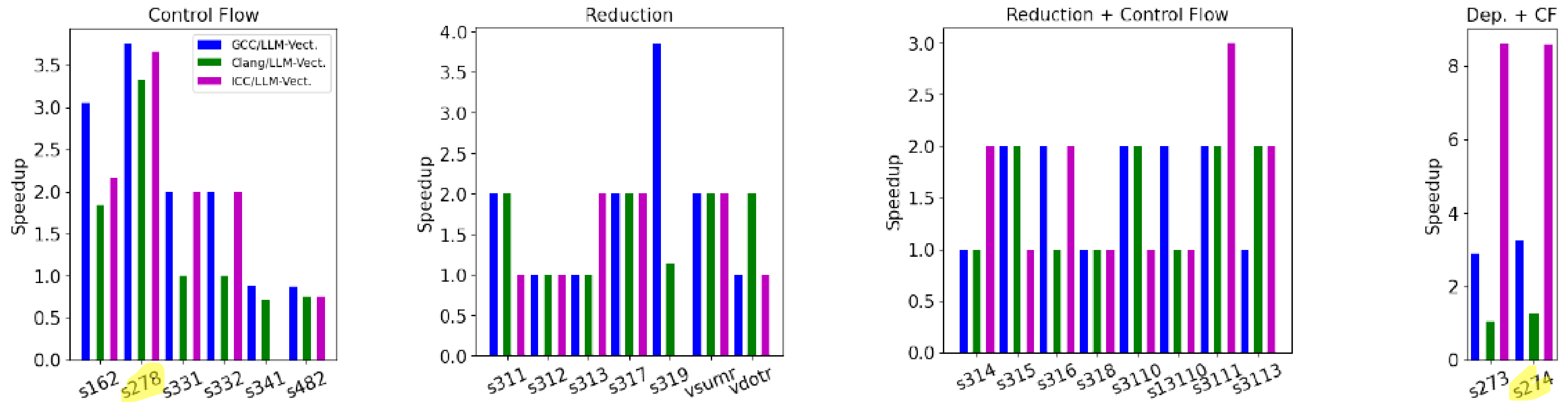
- Bounded loop verification
- Loop Alignment by unrolling unvectorized and vectorized loops
- Unrolling loops at LLVM IR level leads to larger solver queries and solver returns inconclusive results
- Scaling Techniques:
  - C-level unrolling
  - Spatial-case splitting

[RQ] How well can GPT4 vectorize the code on its own?

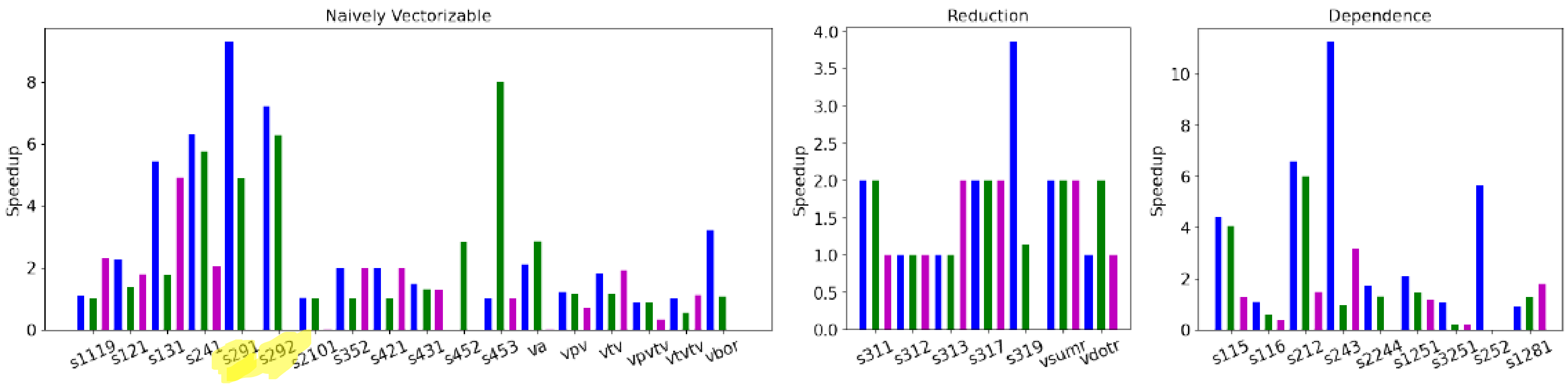
Techniques	Total tests	Equivalent	Not Equivalent	Inconclusive
Checksum-based Testing	149	0	24	125 **[Plausible]**
Alive2	125	26	17	82
C-level Unroll	82	28	18	36
Splitting	36	3	2	31
All	149	57	61	31

38.2% tests proven equivalent

[RQ] Is LLM-vectorized code faster?



(a) Control Flow (b) Reduction (c) Reduction + Control Flow (d) Dep. + CF



(e) Naïvely Vectorizable (f) Reduction (g) Dependence

When LLMs fail?

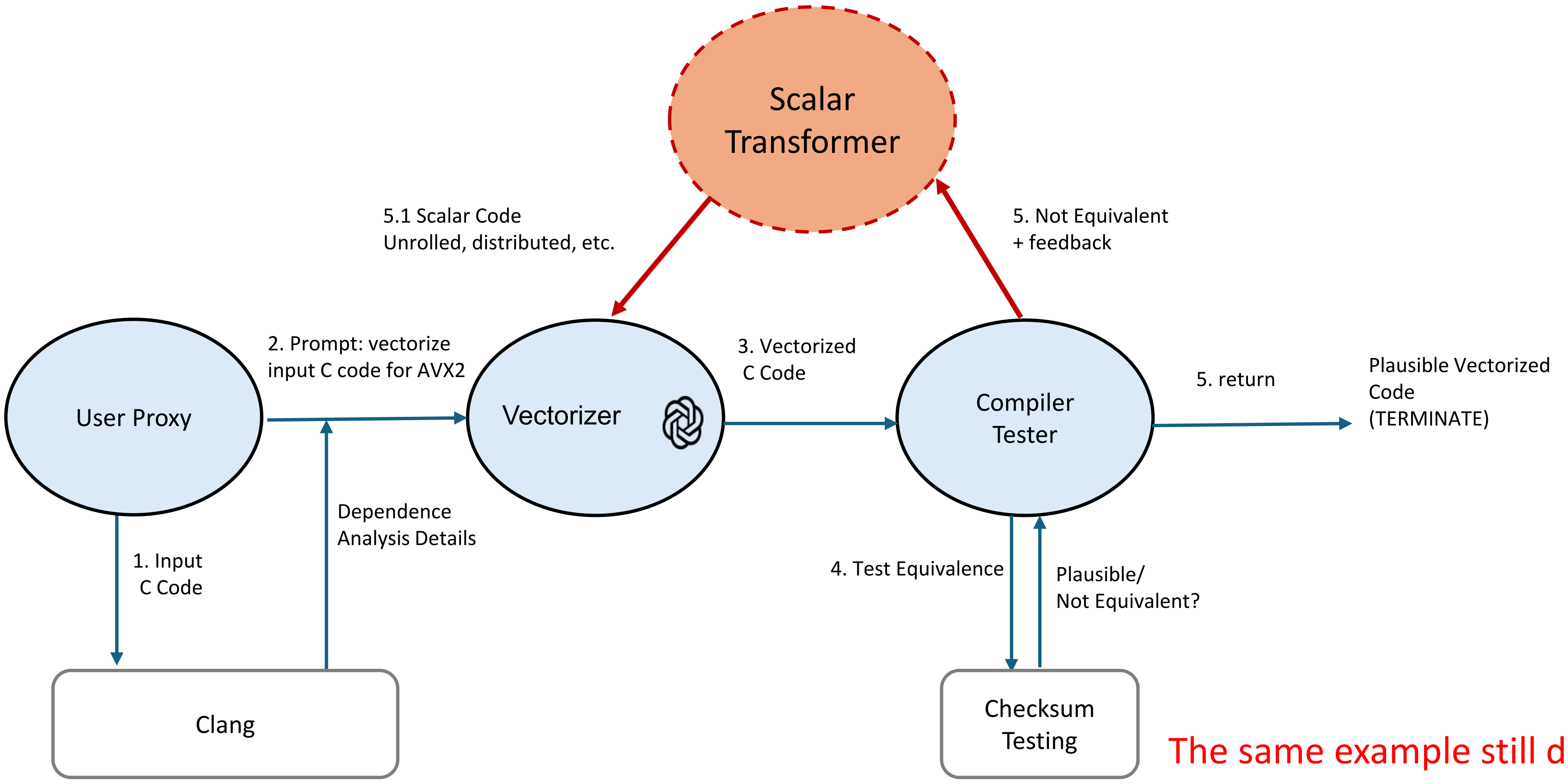
```
void src7(int *a, int *b, int n) {
	for (int i = 0; i < n; i++) {
		a[i] = a[n/2] + b[i];
	}
}
void src7(int *a, int *b, int n) {
    __m256i va_half = _mm256_set1_epi32(a[n / 2]);
    for (int I = 0; i <= n - 8; i += 8) {
        __m256i vb = _mm256_loadu_si256((__m256i *)(b + i));
        __m256i vres = _mm256_add_epi32(va_half, vb);
        _mm256_storeu_si256((__m256i *)(a + i), vres);
    }
}
```

When LLMs fail?

```
void src7(int *a, int *b, int n) {
        for (int i = 0; i < n; i++) {
            a[i] = a[n/2] + b[i];
        }
}
```

```
a[0] = a[4] + b[0]
a[1] = a[4] + b[1]
a[2] = a[4] + b[2]
a[3] = a[4] + b[3]
a[4] = a[4] + b[4]
a[5] = a[4] + b[5]
a[6] = a[4] + b[6]
a[7] = a[4] + b[7]
```

Repair with Scalar Transformation Agent



The same example still doesn't work

Conclusion

Problem:

Auto-vectorization in modern compilers remains conservative.

Our approach:

Leverage LLMs (GPT-4, no fine-tuning) and Formal Verification to generate correct vectorized code.

Conclusion

What we achieved?

LLMs produced vectorized code with up to **9.4× speedups** over compilers, while formal verification (Alive2) ensured correctness for **38.2%** of the tests.

Next Steps:
Unbounded loop verification remains an open problem.
Feedback loop from symbolic verification

Conclusion

Rethinking Compiler Design:
Blending LLMs (AI agents and feedback loops), Formal Methods, Synthesis and human intuition? Thoughts?

~~Can LLMs generate fast code?~~
How can we guarantee LLM-based transformations are both fast and correct?

Email: jubitaneja@microsoft.com
Twitter, GitHub, LinkedIn: @jubitaneja

Backup

Successful Repair Experience

```
//Input code

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, incorrect 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);
}
}
```

Repair Experience

```
//Input code

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];
}
}
```

```
// Second Attempt, plausible  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);
}
}
```

[RQ] Does AI-agents design help?

1.Can we reduce the number of LLM invocations?

2. Can we repair the programs using auxiliary tools (like clang's dependence information) and agents in LLM-Vectorizer?

[RQ] Number of LLM invocations

- For k=1 (one code completion), previously we found 72 plausible tests. With AI-agents, we found 24 new tests => 96 plausible tests with k=1
- With AI-agents, we found 4 unique tests for which 100 code completions (k=100) couldn't find a plausible solution, but agents finds it in k=1.