Finding Missing Automatic Vectorization Opportunity by Differential Testing

東京大学 大学院情報理工学系研究科 コンピュータ科学専攻
高前田研究室 浅野 光平

本発表の概要

☀ 本研究のGoal 🎯

コンパイラの**Vectorizer(ベクトル拡張命令による最適化)**の**バグ/最適化機会**の**自動探索器**を開発し**高性能な汎用コンパイラ基盤**作りに貢献

☀ 本研究のContribution ⛏

- **LoopVectorization**の**バグ/最適化機会自動探索器**のための
  1) **テスト生成**, 2) **差分テスト**
  の効果的な手法を提案
  - Cコンパイラ**Clang**で実験を行い評価

アウトライン

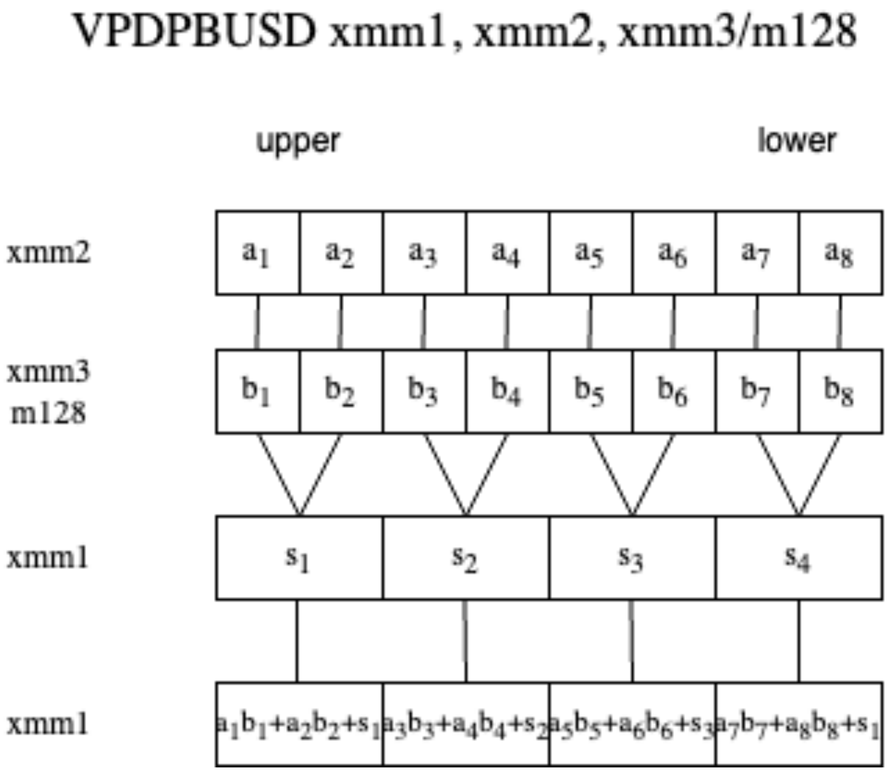
- 背景
  - ベクトル拡張命令/自動ベクトル化/差分テストを用いたコンパイラFuzzing
- 提案手法
  - Vectorizer Fuzzer
  - 評価
- 関連研究, まとめ/今後の展望

アウトライン

- ☀ 背景
  - ▶ ベクトル拡張命令/自動ベクトル化/差分テストを用いたコンパイラFuzzing
- ☀ 提案手法
  - ▶ Vectorizer Fuzzer
  - ▶ 評価
- ☀ 関連研究, まとめ/今後の展望

ベクトル拡張命令

- 1命令で複数の値を処理できるCPUの拡張命令 ≒ SIMD命令(Flynnの分類)
  - ドメイン特化した高抽象度な命令が登場
    - ニューラルネットワーク用のドット積
    - 誤り訂正(CRC)や暗号計算(AES)



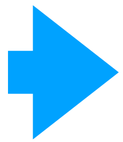
x64 AVX512 VNNIのドット積命令(128bit)

ベクトル拡張命令のポテンシャル

☀ ドット積のCプログラムをコンパイル

gcc -O3 -march=native …

```
void dot_16x1x16_uint8_int8_int32(uint8_t data[4],
        int8_t kernel[16][4],
        int32_t output[16]){
    for (int i = 0; i < 16; i++){
        output[i] = 0;
        for (int k = 0; k < 4; k++){
            output[i] += data[k] * kernel[i][k];
        }
    }
}
```



```
dot_16x1x16_uint8_int8_int32:
        leaq    64(%rdx), %r8
        movq    %rsi, %rcx
        leaq    4(%rdi), %rsi
        cmpq    %r8, %rdi
        setnb   %al
        cmpq    %rsi, %rdx
        setnb   %sil
        orb     %sil, %al
        je      .L2
        leaq    63(%rdx), %rax
        subq    %rcx, %rax
        cmpq    $126, %rax
        jbe     .L2
        movzbl  (%rdi), %eax
        movdqa  .LC0(%rip), %xmm1
        movd    %eax, %xmm0
        movzbl  1(%rdi), %eax
        movdqa  %xmm1, %xmm4
        movdqa  %xmm1, %xmm10
        punpcklbw       %xmm0, %xmm0
        punpcklwd       %xmm0, %xmm0
        pshufd  $0, %xmm0, %xmm6
        movd    %eax, %xmm0
        movzbl  2(%rdi), %eax
        punpcklbw       %xmm0, %xmm0
        movdqa  %xmm6, %xmm11
        punpcklwd       %xmm0, %xmm0
        pshufd  $0, %xmm0, %xmm8
        movd    %eax, %xmm0
        movzbl  3(%rdi), %eax
        punpcklbw       %xmm0, %xmm0
        punpcklwd       %xmm0, %xmm0
        pshufd  $0, %xmm0, %xmm7
        movd    %eax, %xmm0
        punpcklbw       %xmm0, %xmm0
        punpcklwd       %xmm0, %xmm0
        pshufd  $0, %xmm0, %xmm5
        pxor    %xmm0, %xmm0
        movups  %xmm0, (%rdx)
        movups  %xmm0, 16(%rdx)
        movups  %xmm0, 32(%rdx)
        movups  %xmm0, 48(%rdx)
        movdqu  (%rcx), %xmm0
        movdqu  16(%rcx), %xmm3
        movdqu  32(%rcx), %xmm2
        movdqu  48(%rcx), %xmm9
        pand    %xmm3, %xmm10
        psrlw   $8, %xmm3
        pand    %xmm0, %xmm4
        psrlw   $8, %xmm0
        packuswb        %xmm10, %xmm4
        movdqa  %xmm1, %xmm10
        packuswb        %xmm3, %xmm0
        movdqa  %xmm1, %xmm3
        pand    %xmm9, %xmm10
        pand    %xmm2, %xmm3
        psrlw   $8, %xmm9
        psrlw   $8, %xmm2
        packuswb        %xmm10, %xmm3
        movdqa  %xmm1, %xmm10
        packuswb        %xmm9, %xmm2
        movdqa  %xmm1, %xmm9
        pand    %xmm4, %xmm10
        pand    %xmm3, %xmm9
        psrlw   $8, %xmm4
        psrlw   $8, %xmm3
        packuswb        %xmm9, %xmm10
        packuswb        %xmm3, %xmm4
        movdqa  %xmm1, %xmm3
        pand    %xmm2, %xmm1
        pand    %xmm0, %xmm3
```

https://godbolt.org/z/ed1jncsTd

ベクトル拡張命令のポテンシャル

☀ ドット積のCプログラムをコンパイル

手でAVX512-VNNIを使う

```
void dot_16x1x16_uint8_int8_int32(uint8_t data[4],
          int8_t kernel[16][4],
          int32_t output[16]){
    for (int i = 0; i < 16; i++){
          output[i] = 0;
          for (int k = 0; k < 4; k++){
              output[i] += data[k] * kernel[i][k];
          }
    }
}
```

```
dot_16x1x16_uint8_int8_int32:
    vmovdqu64 zmm0, (%rdx)
    vpbroadcastd zmm1, (%rdi)
    vpdpbusd zmm0, zmm0, (%rsi)
    vmovdqu64 (%rdx), zmm0
    ret
```

1/35サイズ

11倍 高速化

ベクトル命令の活用度合いは効率的な計算に大きく影響↗️

ベクトル拡張命令の活用方法

高抽象度

- コンパイラの**自動ベクトル化(Vectorizer)**を使う **(本研究の対象)**

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
```

- コンパイラのディレクティブを書く

- Intrinsicsを書く

```
__m128i _mm_dpbusd_epi32(__m128i, __m128i, __m128i);
__m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
```

- アセンブリを書く

```
    vmovdqu64 zmm0, (%rdx)
    vpbroadcastd zmm1, (%rdi)
    vpdpbusd zmm0, zmm0, (%rsi)
    vmovdqu64 (%rdx), zmm0
```

低抽象度

自動ベクトル化アルゴリズム: Overview

☀ 現代の**汎用言語**のコンパイラの**自動ベクトル化**アルゴリズム

1. SLP Vectorization [Samuel+, PLDI '20]
2. Loop Vectorization
3. Ad hocなPeephole Optimization

自動ベクトル化アルゴリズム: 1, 2

- レジスタの個数/サイズ, キャッシュ, 命令実行速度(コストモデル)など
  を用いる

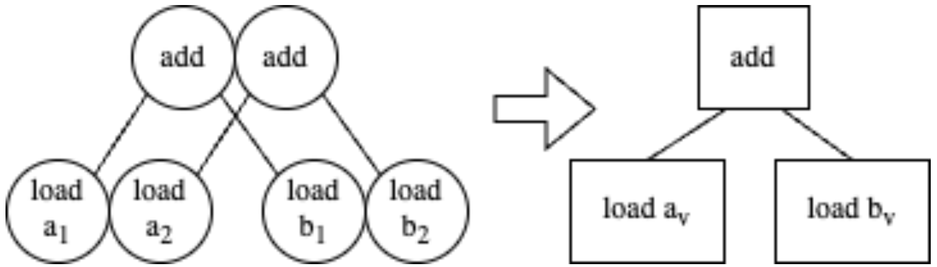
- **SLP(Superword-Level Parallelism)Vectorization** [Samuel+, PLDI '20]

  - ストレートなコードのベクトル化

- **LoopVectorization(LV)**

  - Loop単位のベクトル化

- それぞれLegality, Profitability Checkの後
  対象とする構造をベクトル化



SLPVectorization

自動ベクトル化アルゴリズム: 3

☀ それぞれの命令セットに固有のPeephole Optimization

```
vpmaddwd ymm3,ymm3,YMMWORD PTR [r8+0x20]
vpaddd ymm2,ymm2,ymm3
```

⇒

```
vpdpwssd ymm2,ymm3,YMMWORD PTR [r8+0x20]
```

[X86] Combine reduce (add (mul x, y)) to VNNI instruction.

Closed Public

[X86] Machine combine vnni instruction.

Closed Public

Authored by **LuoYuanke** on Apr 22 2023, 1:39 PM.

[AArch64] turn extended vecreduce bigger than v16i8 into udot/sdot

Closed Public

[AArch64] Generate dot for v16i8 sum reduction to i32

Closed Public

Authored by **mivnay** on Oct 1 2020, 12:24 AM.

各ターゲットごとに個別に実装

Vectorizerの管理/開発の困難さ

- ☀ 他の最適化と比べ, ハードウェアに依存しているため管理が困難
  - ▶ 仕様の変化への対応
    - **可変長ベクトル(Arm Scalable Vector Extension**, RISC-V RVV)
    - コストモデル
    - 命令セット

Vectorizerの開発コスト削減の先行研究

- VeGen: A Vectorizer Generator for SIMD and Beyond

  [Chen+, ASPLOS '21]

  - 仕様からLLVMのad-hoc peephole optimizerを生成するVectorizer Generatorを実装
  - 可変長ベクトルは非対応, 既存の実装との親和性が低い

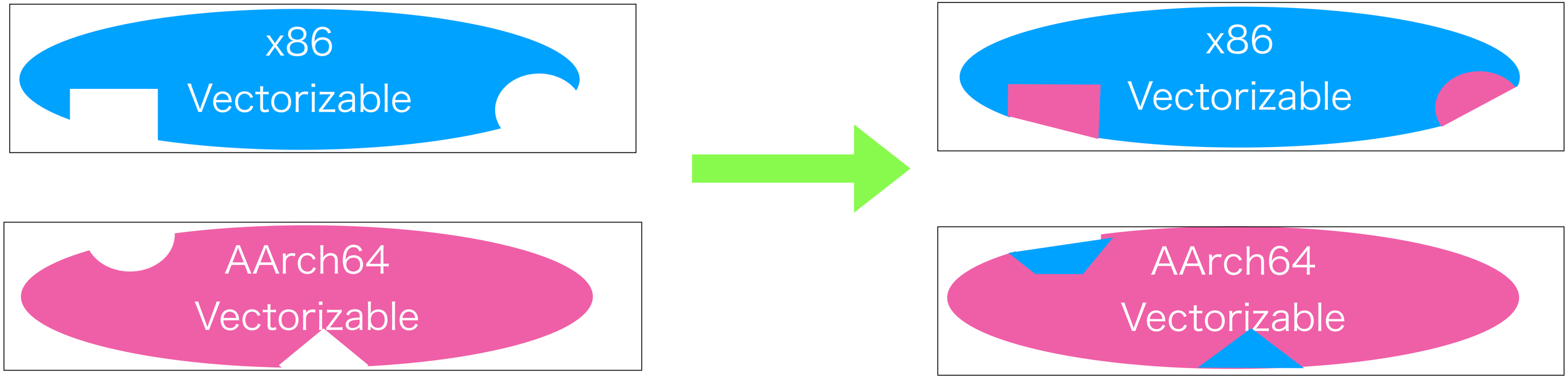
- Minotaur: A SIMD-Oriented Synthesizing Superoptimizer

  [Zhengyang+, arxiv '23]

  - SIMD IntrinsicsでのAd-hoc peephole optimizer機会を自動で探索
  - 可変長ベクトルは非対応, コンパイル時間が膨大

本研究のアプローチ

仕様の変更に強く，既存のAd-hocな実装と親和性の高い，
異なるターゲット間で**相補的**なVectorizerの最適化機会探索💪



差分テストベースのコンパイラFuzzing

☀ 本研究では,ブラックボックスな**差分テストベースのコンパイラFuzzing**を用いる

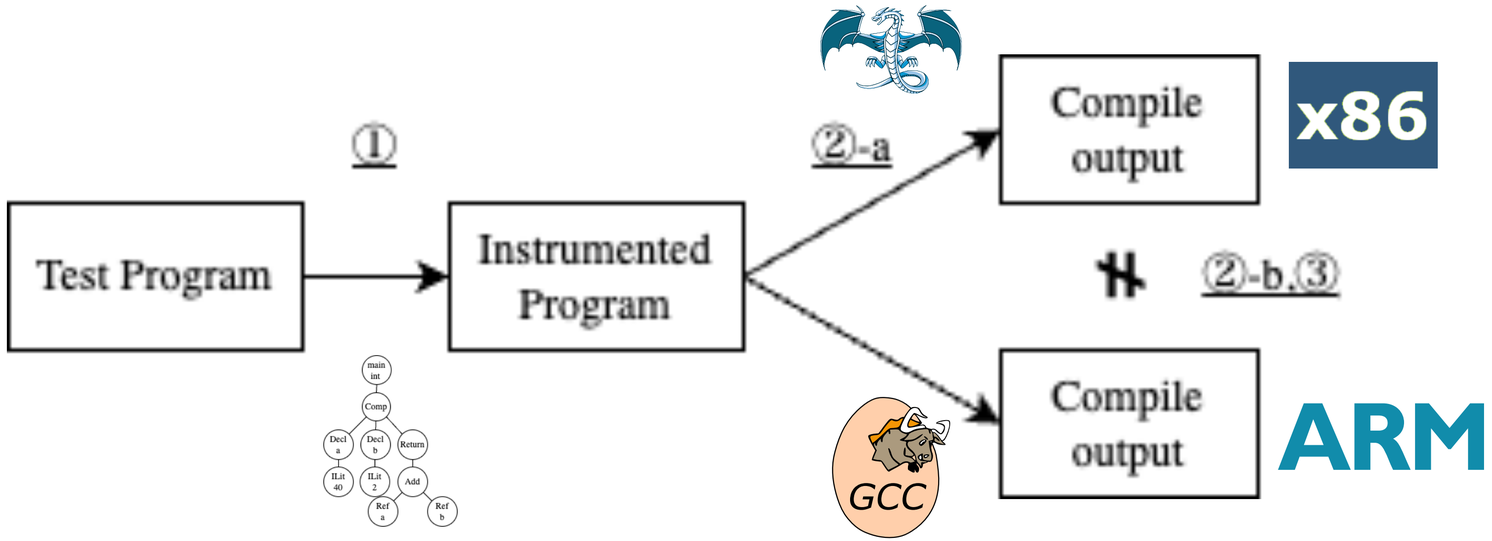
① テスト生成

② 差分テスト

a. コンパイル

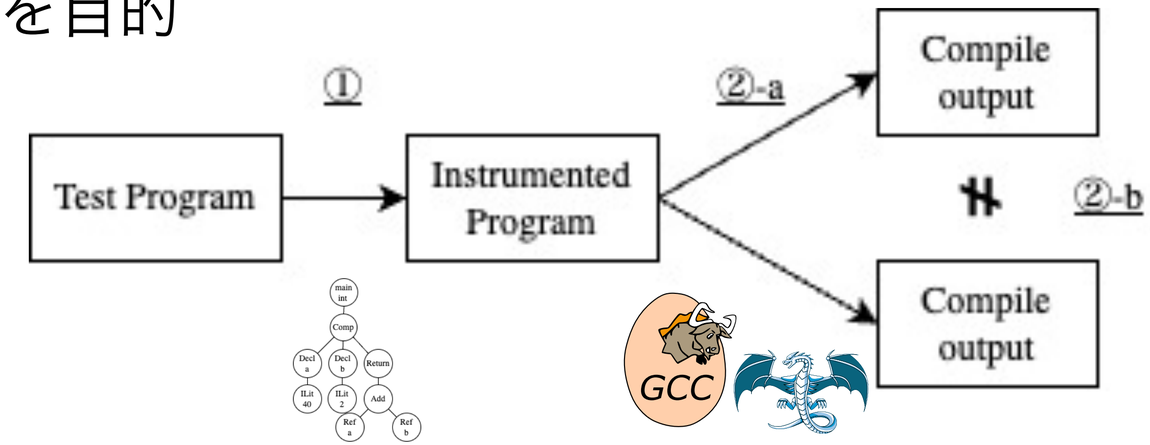
b. 比較

③ 解析, レポート



関連研究: 差分テストベースのコンパイラFuzzing

☀ **複雑な最適化に対するバグや機会の自動報告**を目的



- UBSanitizerのFalse Negativeの自動探索 [Shaohua+, ASPLOS '24]

①… UB 挿入,　　②-a…UBSan + -O3/-O0　　②-b…UBSanの検知比較

- LLVM IR最適化Regressionの自動探索 [Theodoros+, ASPLOS '22]

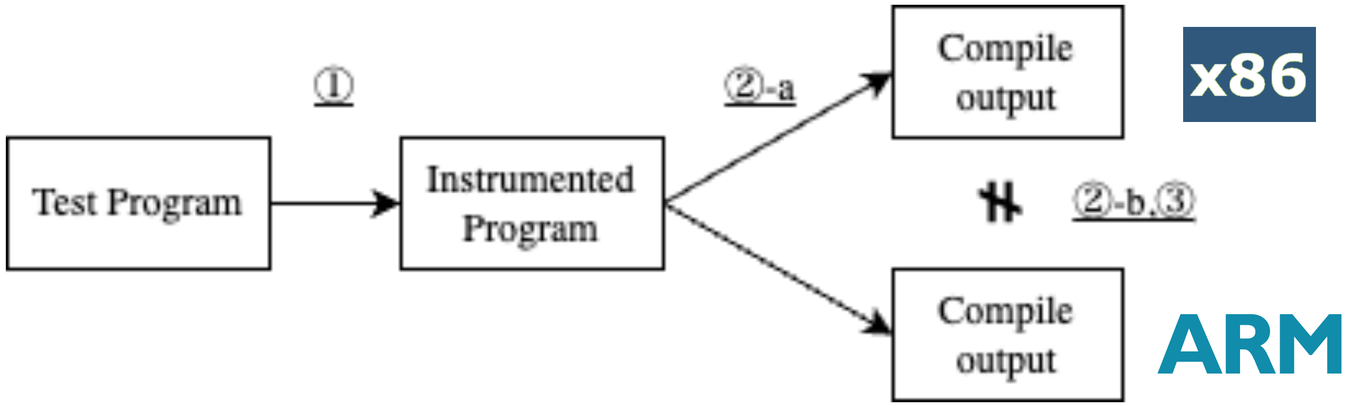
①… Markerマクロ挿入　　②-a …-O3/-O0, Clang/GCC　②-b …Markerの存在比較

アウトライン

- 背景
  - ベクトル拡張命令/自動ベクトル化/差分テストを用いたコンパイラFuzzing
- 提案手法
  - Vectorizer Fuzzer
  - 評価
- 関連研究, まとめ/今後の展望

提案: 差分テストベースのVectorizer Fuzzing

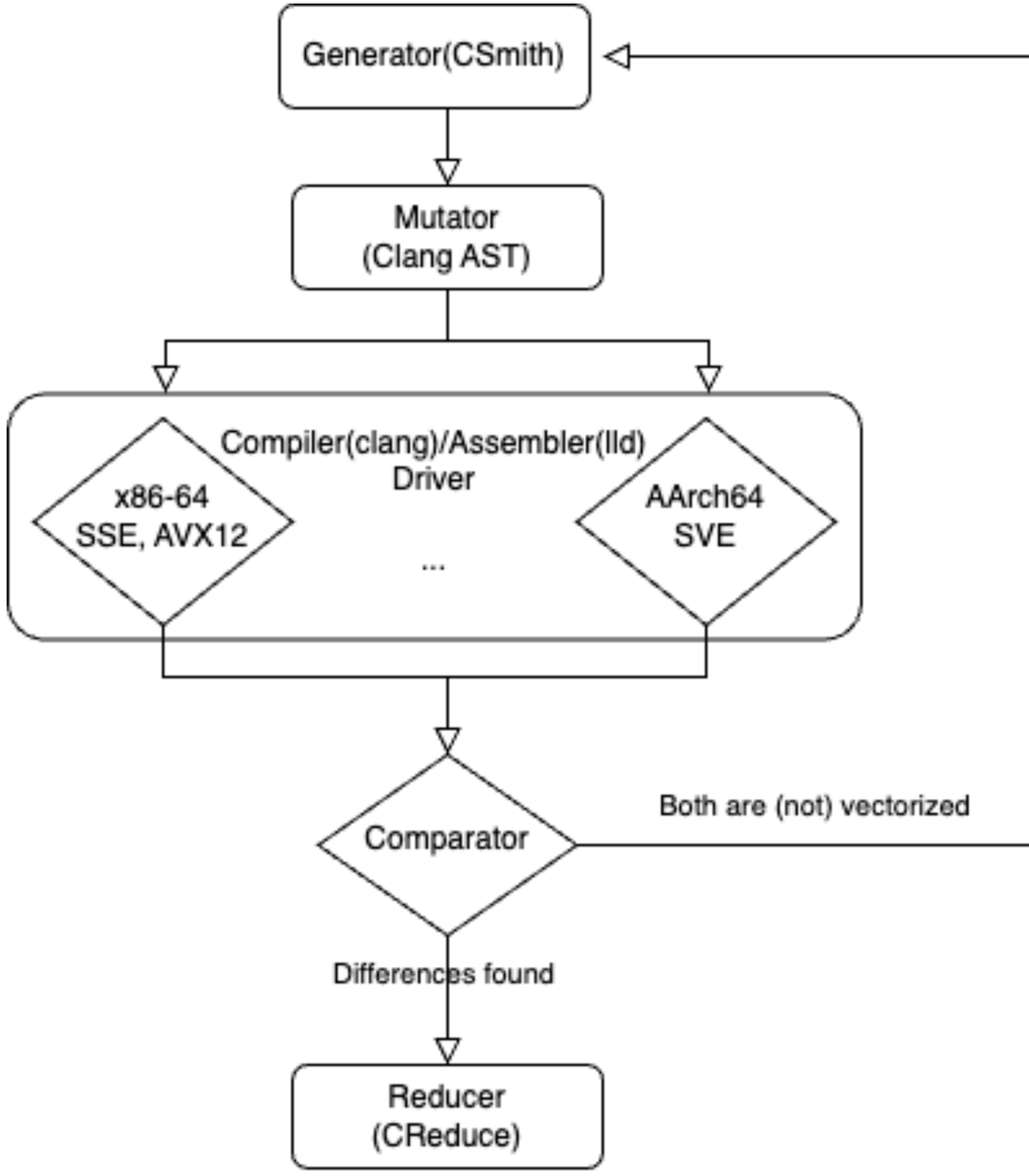
☀ 本研究では**Loop構造にフォーカス**した差分テストベースFuzzingのための
①テスト生成, ②差分テスト 手法を提案



Vectorizerは各Targetへ依存しているため
差分テストの設計がチャレンジ🔥

提案: Vectorizer Fuzzer

1. **Generator** … ベースプログラムの生成
2. **Mutator** … **Vectorizer用のテスト**への変更
3. **Compiler(s)** … 各Target向けにコンパイル
4. **Comparator** … **抽象化された値**で比較
5. **Reducer** … 解析 (Future work)

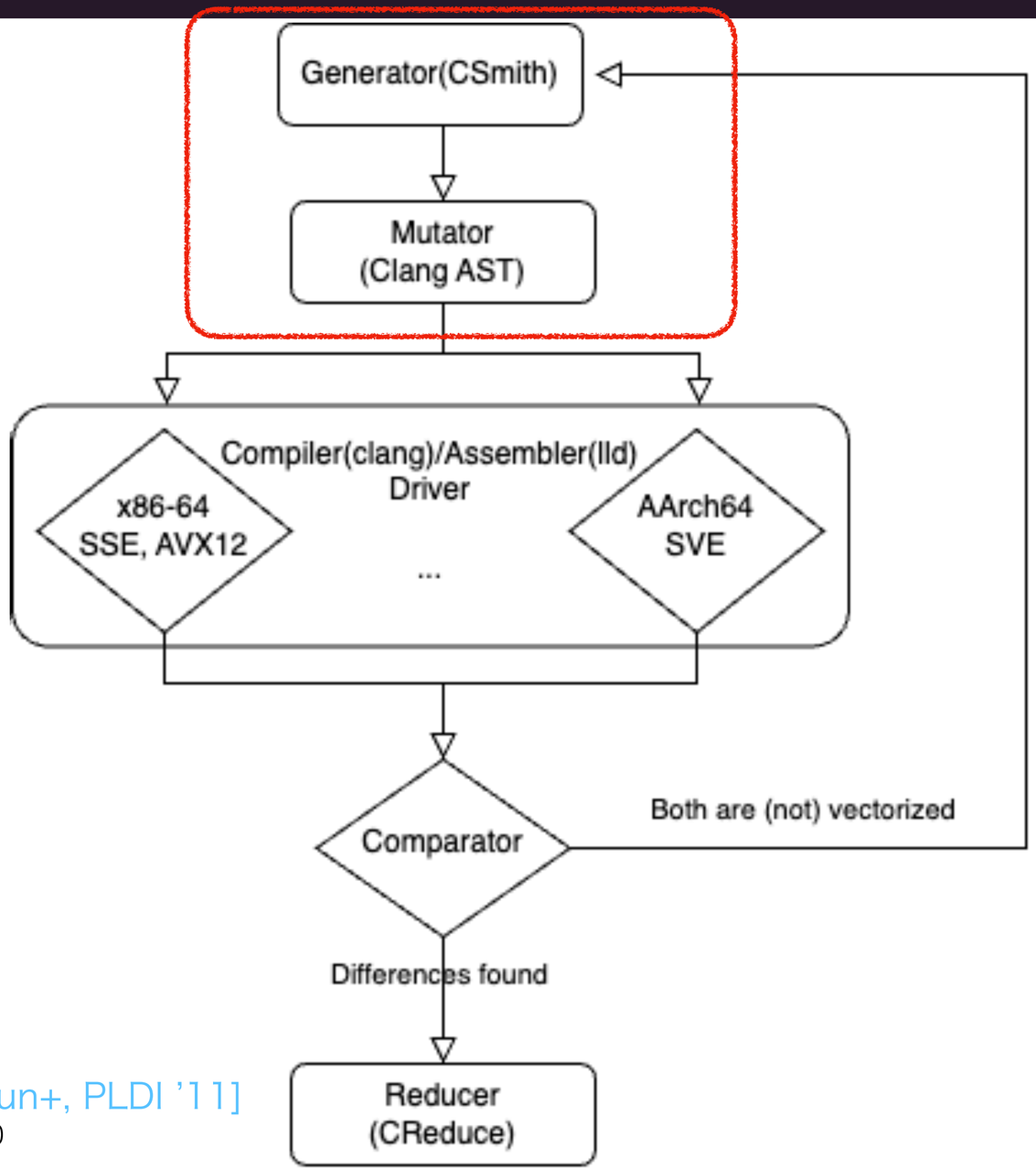


提案: Vectorizer Fuzzer

1. Generator
2. Mutator

テスト生成

3. Compiler(s)
4. Comparator
5. Reducer



CSmith*で生成したループプログラムを
Profitableに書き換え

*Finding and Understanding Bugs in C Compilers [Xuejun+, PLDI '11]

提案: Vectorizer Fuzzer

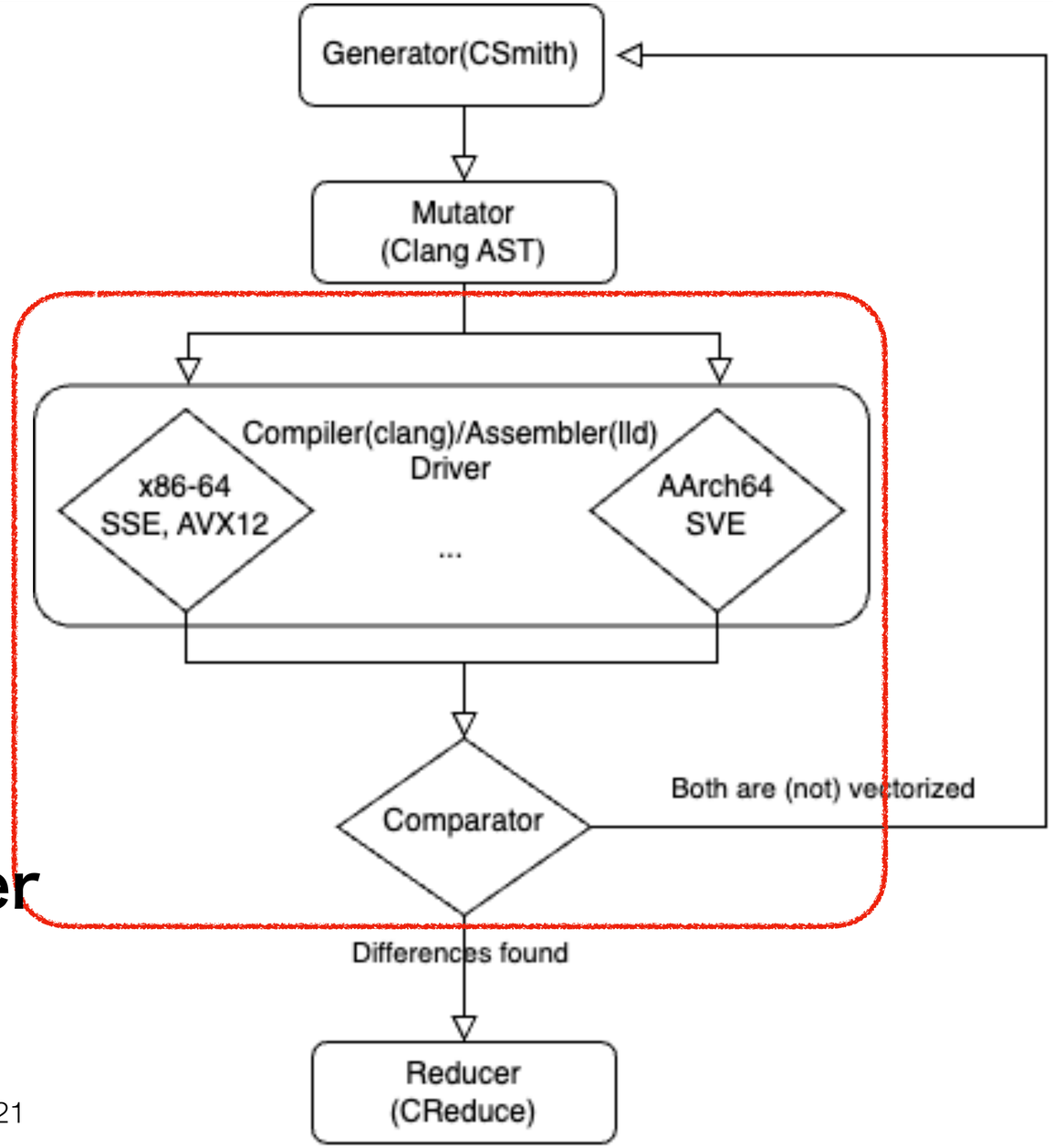
1. Generator
2. Mutator

テスト生成

3. Compiler(s)
4. Comparator

差分テスト

5. Reducer

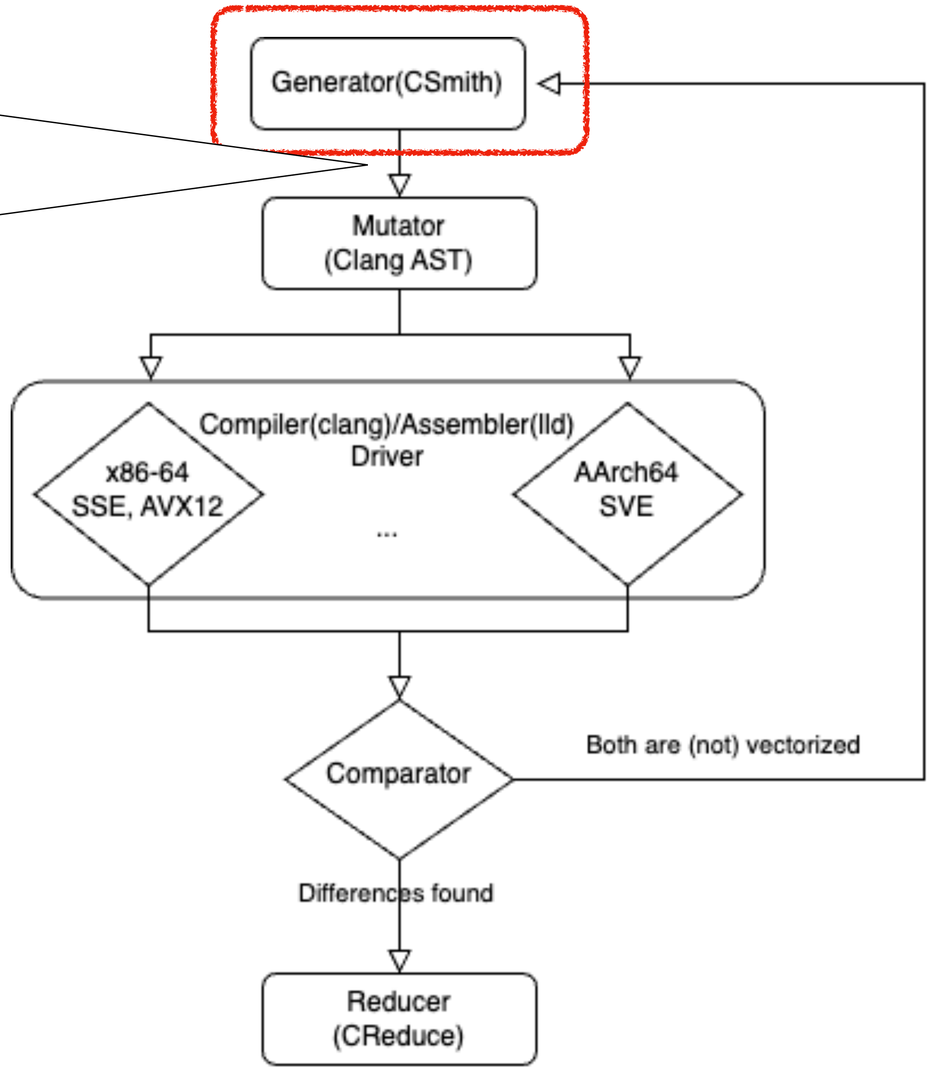


比較はLoopVectorizer, SLPVectorizer のベクトル化統計値により行う

Running Example: Generator

1. **Generator**
2. Mutator
3. Compiler(s)
4. Comparator
5. Reducer

```
…
int8_t ****l_962[10][8][3] =
{{{&g_891,...}}};
int16_t l_1199 = 7L;
const union U0 *l_1240 = &g_247;
uint64_t ***l_1246 = &g_452;
int32_t l_1364 = (-1L);
uint32_t l_1408 = 4294967295UL;
int i, j, k;
for (i = 0; i < 9; i++)
{
    for (j = 0; j < 1; j++)
    {
        for (k = 0; k < 1; k++)
            l_598[i][j][k] = 0UL;
    }
}
for (i = 0; i < 2; i++)
    l_620[i] = &l_621[1][0][2];
…
```

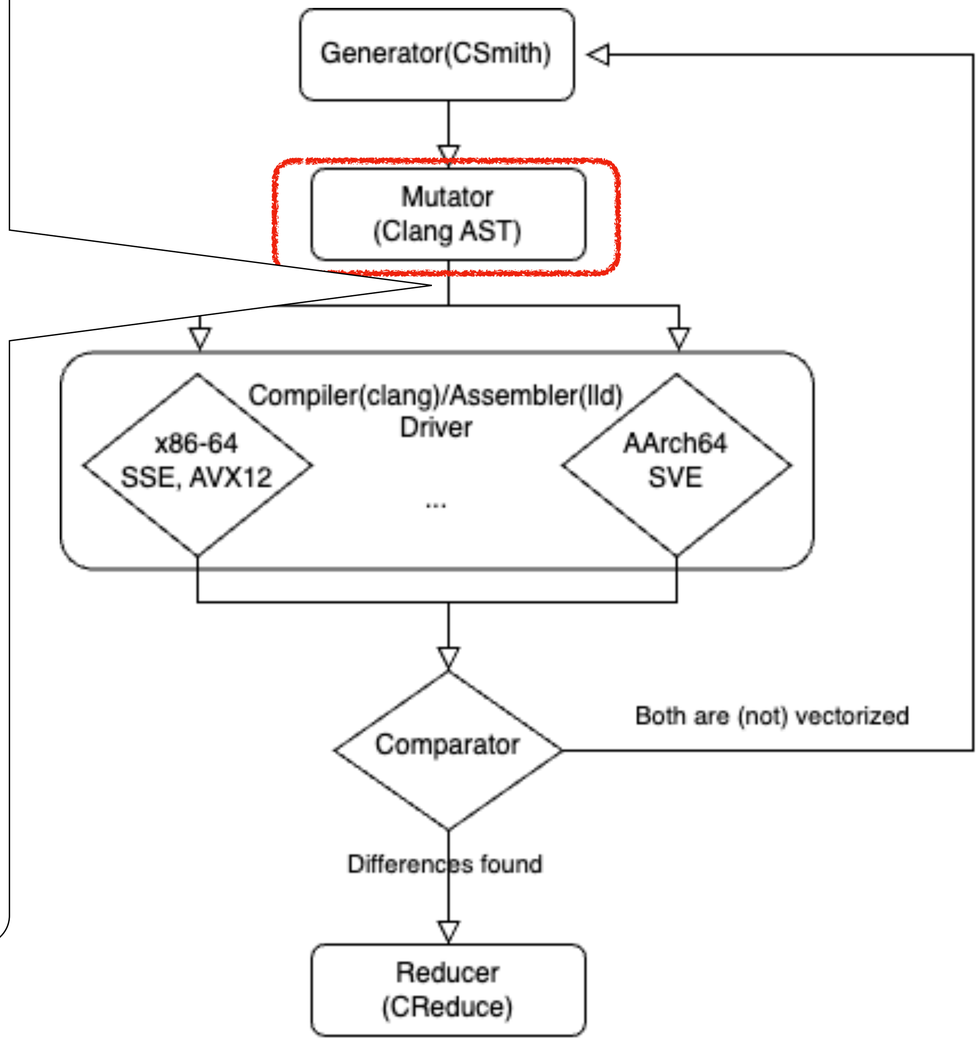


Generated by CSmith

Running Example: Mutator

1. Generator
2. **Mutator**
3. Compiler(s)
4. Comparator
5. Reducer

```
…
alignas(128) int8_t ****l_962[4096]
[4096][4096]
     ={{{&g_891…}};
int16_t l_1199 = 7L;
const union U0 *l_1240 = &g_247;
uint64_t ***l_1246 = &g_452;
int32_t l_1364 = (-1L);
uint32_t l_1408 = 4294967295UL;
int i, j, k;
for (i = 0; i < 4096; i++)
{
    for (j = 0; j < 4096; j++)
    {
        for (k = 0; k < 4096; k++)
            l_598[i][j][k] = 0UL;
    }
}
for (i = 0; i < 4096; i++)
    l_620[i] = &l_621[1][0][2];
```



Running Example: Compiler(s)

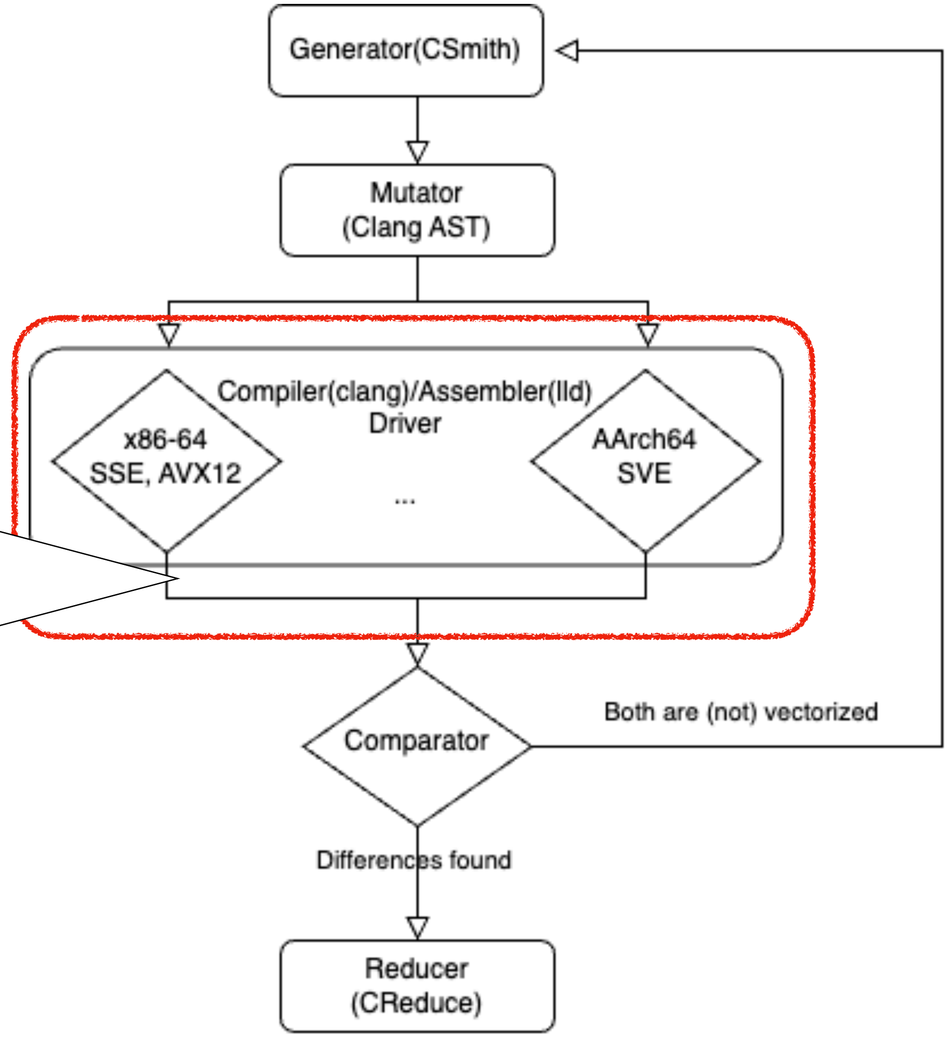
1. Generator
2. Mutator
3. **Compiler(s)**
4. Comparator
5. Reducer

x86-64 + avx512f

```
…
.LBB0_4:
  vptestmd  %zmm1, %zmm0, %k1
  vpsrld  $1, %zmm0, %zmm4
  vpxord  %zmm2, %zmm4, %zmm4 {%k1}
…
```

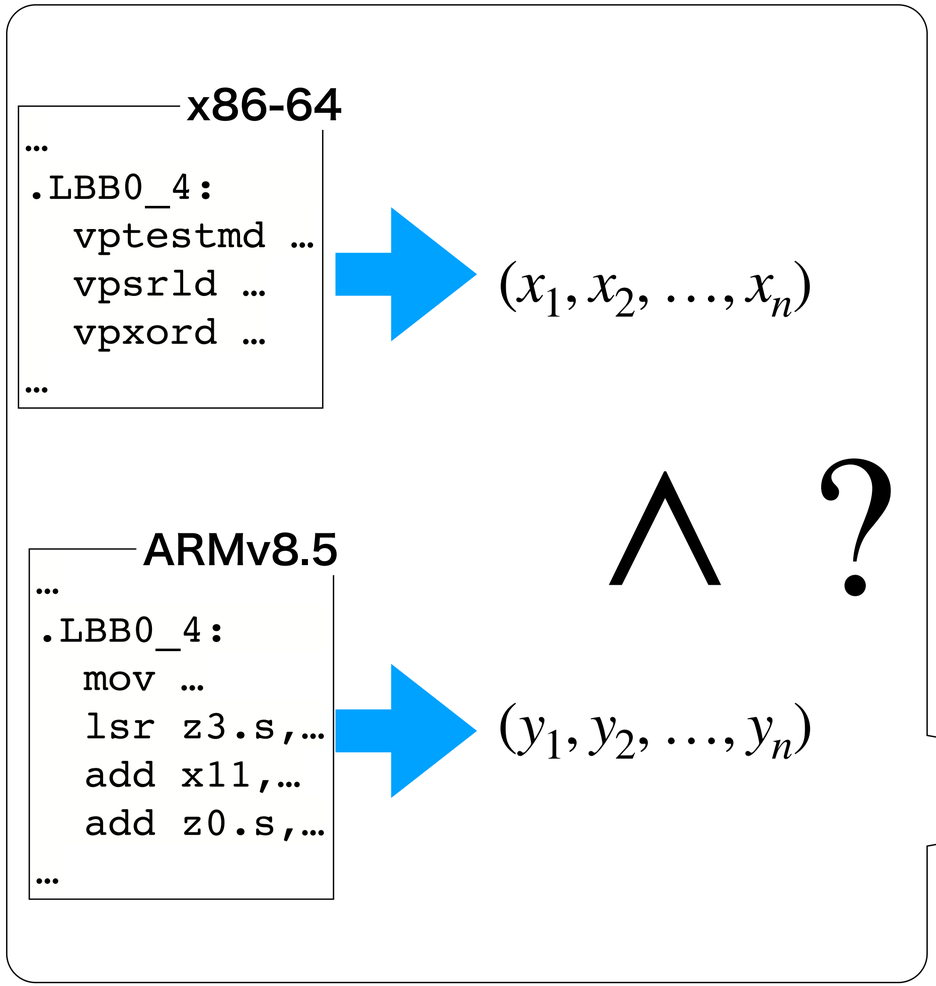
ARMv8.5 + sve
(512 bit register width)

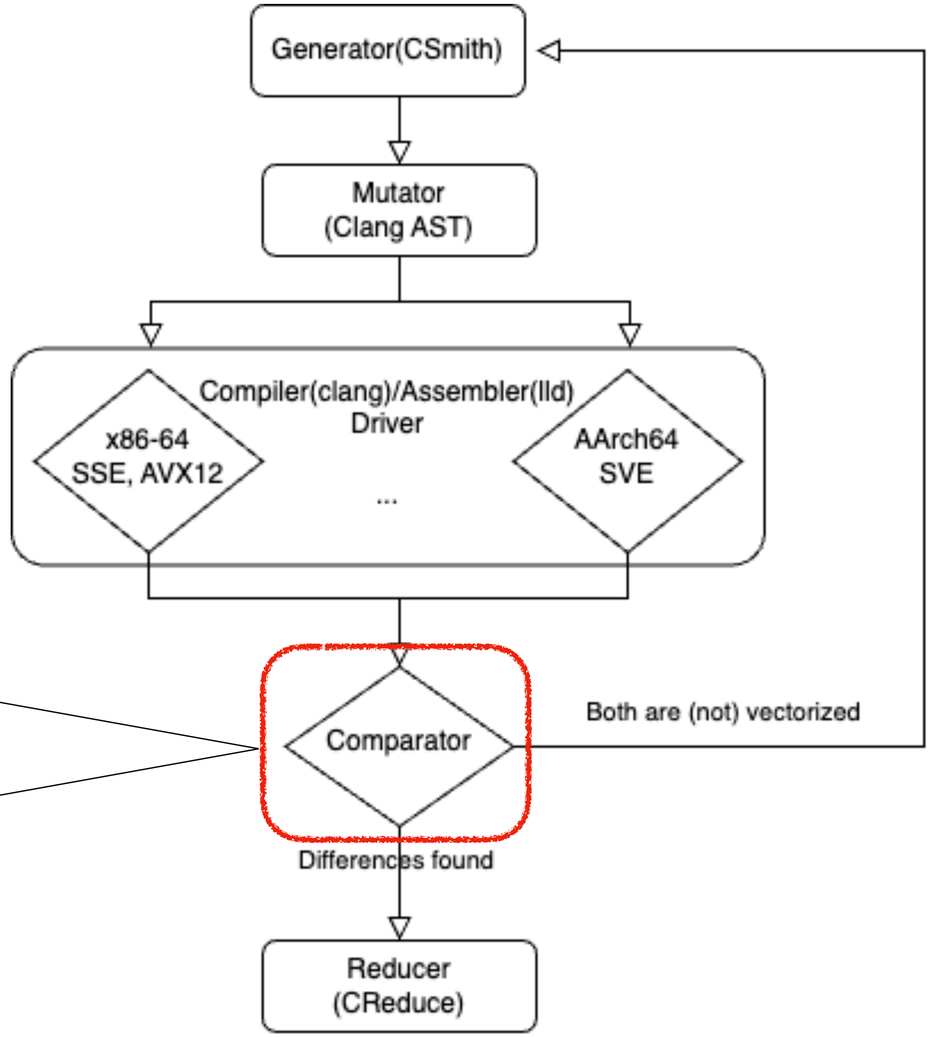
```
…
.LBB0_4:
  mov z2.d, z0.d
  lsr z3.s, z0.s, #1
  add x11, x9, x8, lsl #2
  add z0.s, z0.s, #16
…
```



Running Example: Comparator

1. Generator
2. Mutator
3. Compiler(s)
4. **Comparator**
5. Reducer





Running Example: Reducer(Future work)

1. Generator
2. Mutator
3. Compiler(s)
4. Comparator
5. **Reducer**

Generated C program

```
        int32_t l_516 = (-1L);
        int32_t l_522 = 0x4ACC47C3L;
        alignas(128) int8_t *l_572[4096]
[4096]={{&g_468,&g_468,&g_468,&g_468,&g_468
,&g_468},
{&g_573,&g_468,&g_573,&g_573,&g_468,&g_573}
,
{&g_573,&g_468,&g_468,&g_573,&g_468,&g_573}
,
{&g_573,&g_468,&g_573,&g_573,&g_468,&g_468}
,
{&g_573,&g_468,&g_573,&g_573,&g_468,&g_573}
,
{&g_573,&g_468,&g_468,&g_573,&g_468,&g_573}
};
        alignas(128) int32_t *l_597[4096]
[4096]={{&l_522,&g_147,&g_147,&l_522,&l_522
,&g_147,&g_147,&l_522,&l_522},
{&l_581,&g_126,&l_581,&g_126,&l_581,&g_126,
&l_581,&g_126,&l_581},
{&l_522,&l_522,&g_147,&g_147,&l_522,&l_522,
&g_147,&g_147,&l_522},{(void*)0,&g_126,
(void*)0,&g_126,(void*)0},
{&l_522,&g_147,&g_147,&l_522,&l_522,&g_147,
&g_147,&l_522,&l_522},
{&l_581,&g_126,&l_581,&g_126,&l_581,&g_126,
&l_581,&g_126,&l_581},
{&l_522,&l_522,&g_147,&g_147,&l_522,&l_522,
&g_147,&g_147,&l_522},{(void*)0,&g_126,
(void*)0,&g_126,(void*)0},
{&l_522,&g_147,&g_147,&l_522,&l_522,&g_147,
&g_147,&l_522,&l_522}};
        alignas(128) uint8_t l_598[4096]
[4096];
        uint64_t *l_611 = &g_144[2][1][5];
        alignas(128) const uint32_t
*l_621[4096][4096]
[4096]={{{&l_599,&g_107,&l_599}},
{{&l_599,&g_107,&l_599}}};
        alignas(128) const uint32_t
**l_620[4096];
        int32_t l_671 = 5L;
        union U0 *l_723 = &g_622;
        uint16_t *l_741 = &g_28[5][1][0];
        uint16_t **l_740 = &l_741;
        int32_t ** const * const **l_799 =
(void*)0;
        int64_t l_828 =
0x4C1C55A4356F194BLL;
        uint32_t l_863 = 0x6869705BL;
        int64_t l_905 = (-1L);
        alignas(128) int8_t ****l_962[4096]
[4096][4096]={{{{&g_891,&g_891,&g_891},
{&g_891,&g_891,&g_891},
{&g_891,&g_891,&g_891},
{(void*)0,&g_891,&g_891},{&g_891,(void*)0,
(void*)0},{&g_891,(void*)0,&g_891},{&g_891,
(void*)0,(void*)0},{&g_891,&g_891,
(void*)0}},{{&g_891,&g_891,&g_891},
{&g_891,&g_891,(void*)0},
{&g_891,&g_891,&g_891},
{&g_891,&g_891,&g_891},
{(void*)0,&g_891,&g_891},
{&g_891,&g_891,&g_891},
{&g_891,&g_891,&g_891},
{&g_891,&g_891,&g_891}},
{{{&g_891,&g_891,&g_891},{(void*)0,&g_891,
(void*)0},{&g_891,&g_891,&g_891},
{&g_891,&g_891,&g_891},
{&g_891,&g_891,&g_891},
```

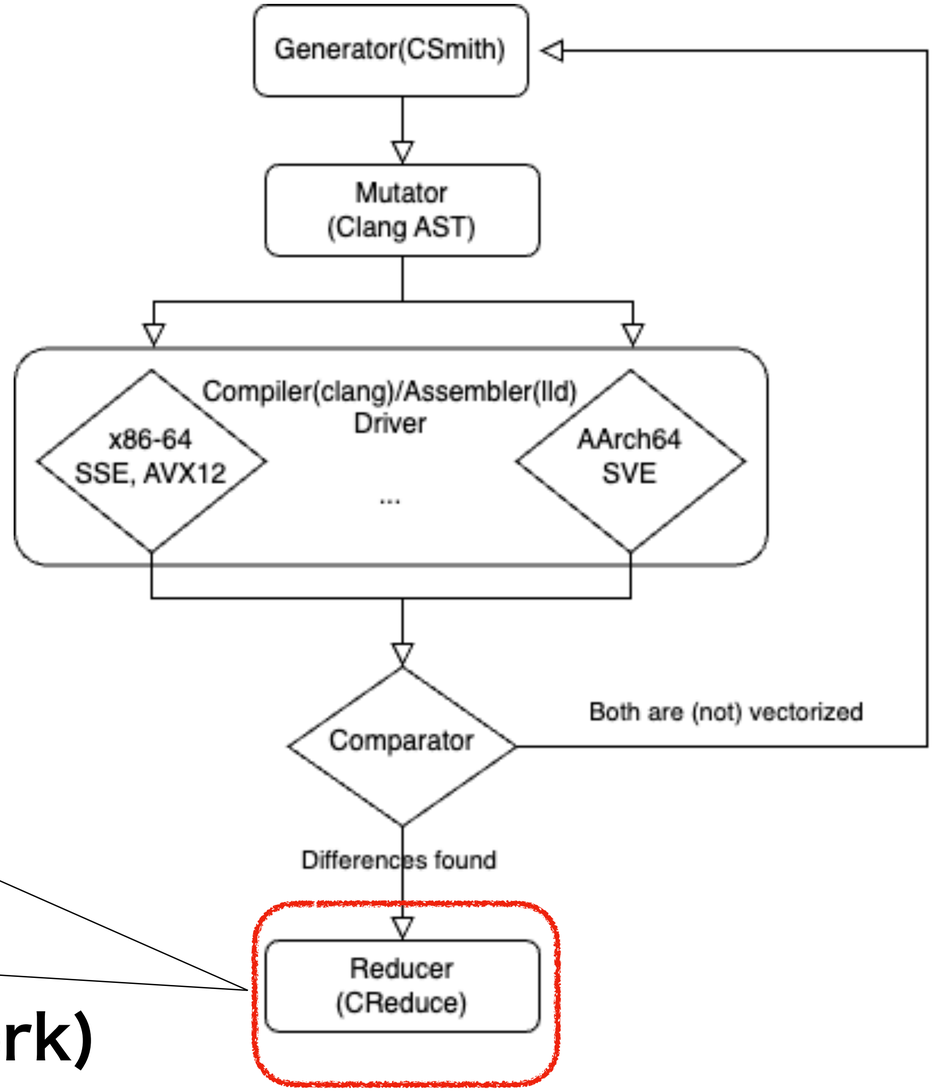
$(x_1, x_2, \ldots, x_n)$

Reduced

```
void main() {
alignas(128) int8_t ****l_962[4096][4096]
[4096]
     ={{{&g_891…}};
int16_t l_1199 = 7L;
const union U0 *l_1240 = &g_247;
uint64_t ***l_1246 = &g_452;
int32_t l_1364 = (-1L);
uint32_t l_1408 = 4294967295UL;
int i, j, k;
for (i = 0; i < 4096; i++)
{
    for (j = 0; j < 4096; j++)
    {
        for (k = 0; k < 4096; k++)
            l_598[i][j][k] = 0UL;
    }
}
for (i = 0; i < 4096; i++)
```

$\wedge$

$(y_1, y_2, \ldots, y_n)$



CReduceで行うのが主流(本研究ではFuture work)

評価

☀ テスト生成器①は**自動ベクトル化のテスト**に**良い**テストを生成できるか

→ LoopVectorizerのVectorization機会を多く埋め込んでいる
(評価1)

☀ 差分テスト②を用いて**Target間のVectorizationの性能が比較できるか**

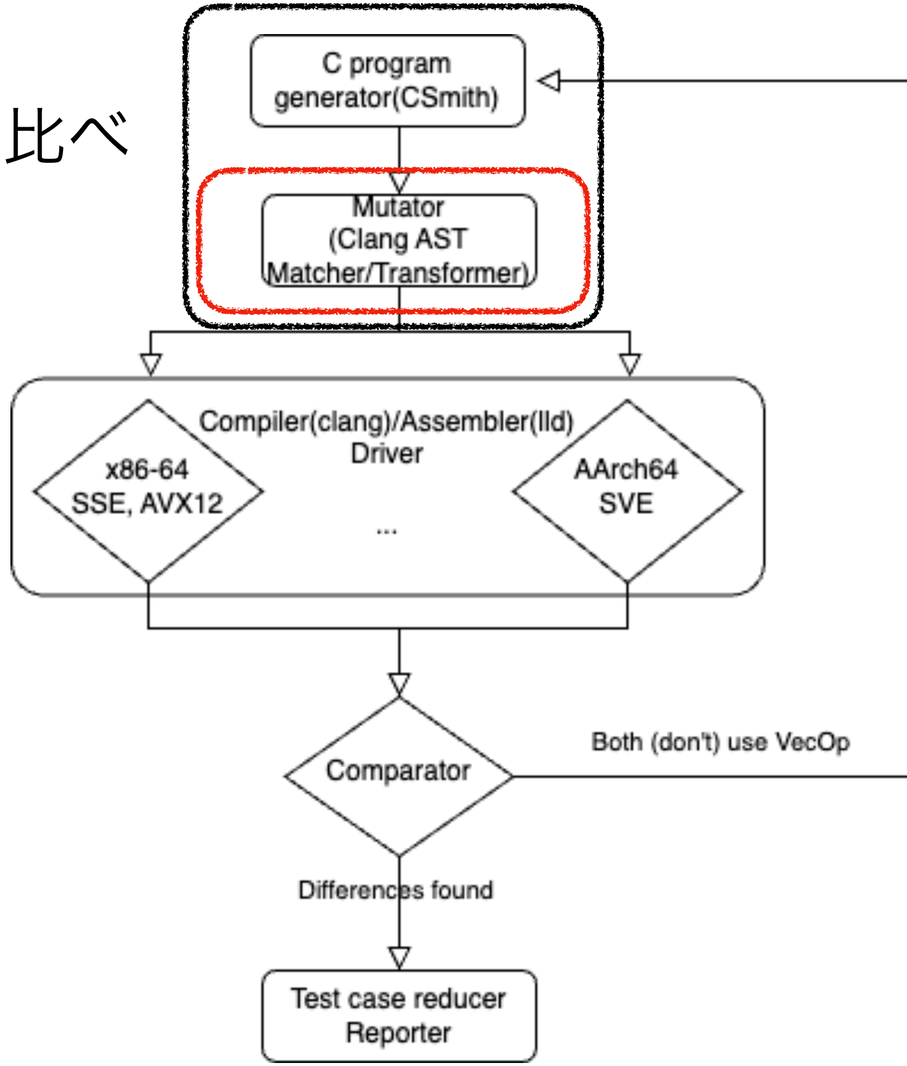
→Arm SVEとx86 AVX512 間のLoopVectorizerの差を発見
(評価2)

評価1: ベクトル化テスト生成能力の検証

☀ **Mutator**適用後のプログラムはCSmithそのままと比べVectorizerに解析される構造を多く含むか
以下の**コンパイル時**統計値を用いて比較

- LoopVectorizeが**ベクトル化*したループ**の数

- SLPVectorizerが**ベクトル化*した命令**数

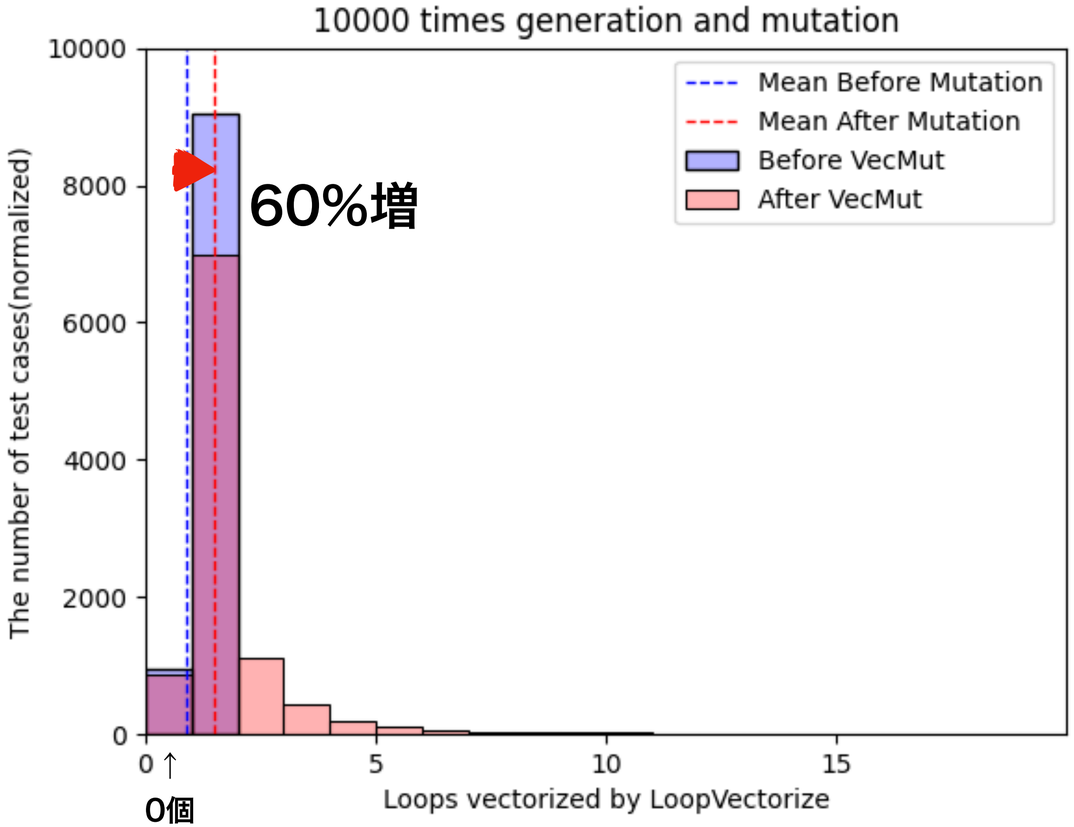


* Legality, Profitability Checkをクリアする
=> ベクトル化される

評価1: ベクトル化テスト生成能力の検証

☀ 10,000回のうちLoopVectorizeが**ベクトル化するループ**の個数

- 1ケースあたりのループ個数は**60%増**
  - **2238**ケースで増加(80ケースが減少)

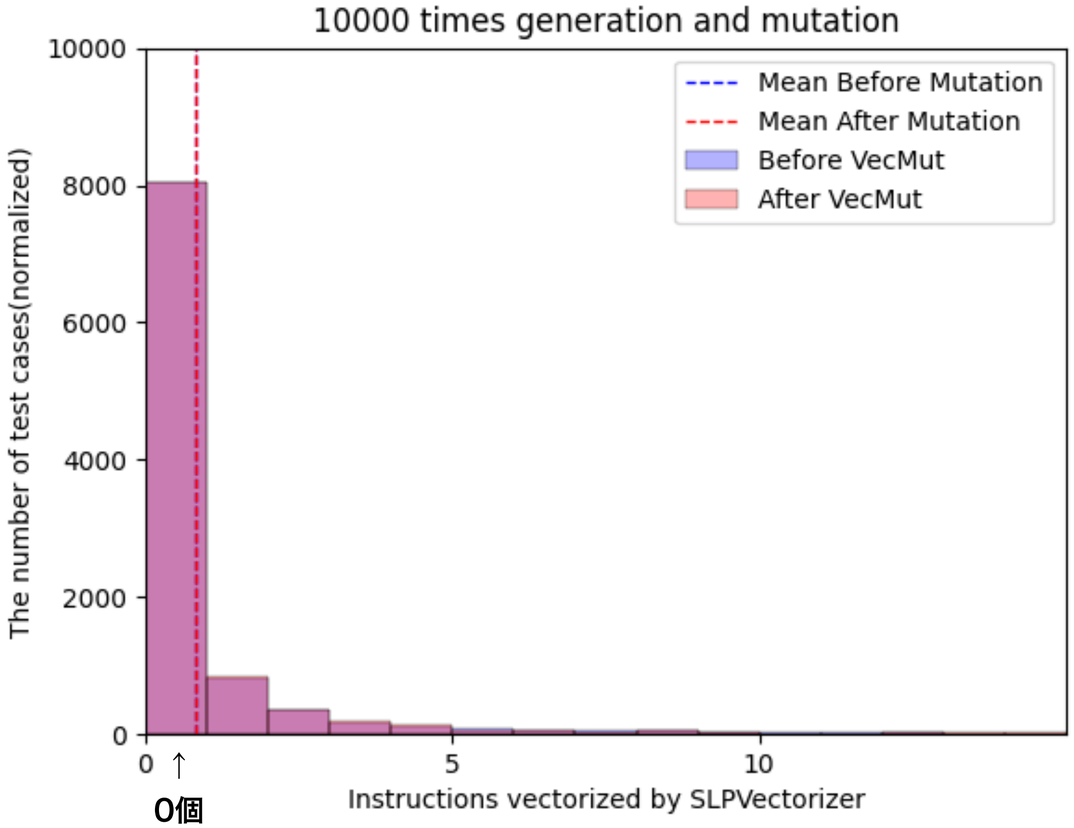


→Loop Vectorizeのベクトル化機会が増加
Legality Checkの回数も増え, (Appendix)
テスト生成としては効果的

評価1: ベクトル化テスト生成能力の検証

☀ 10000回のうちSLPVectorizerが**ベクトル化する命令**の個数

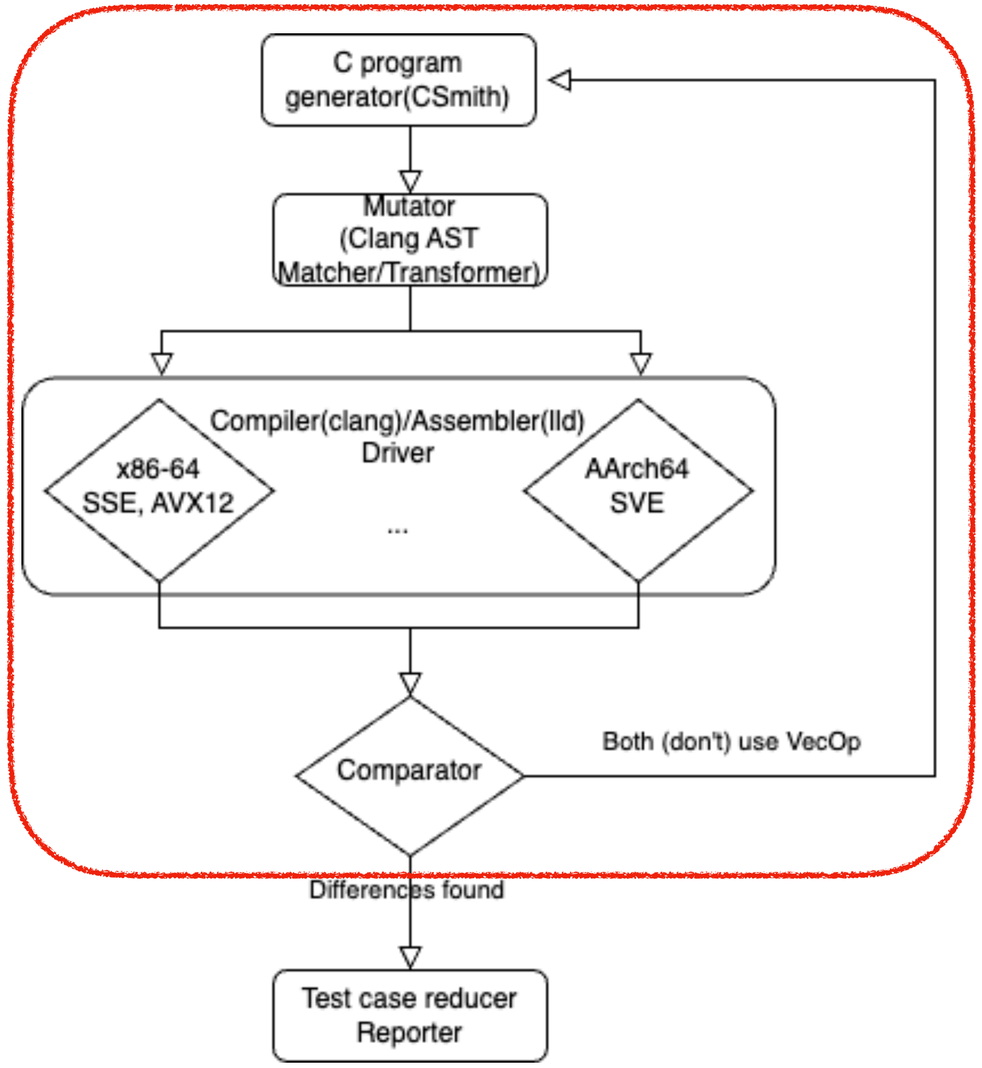
- 1ケースあたりの命令個数は3%増
  - 554ケースで増加(527ケースで減少)



→SLPの機会の個数はほぼ不変
(同型な命令列に関する操作はしてないため)

評価2: SVEとAVX512の自動ベクトル化の性能差発見

- **可変/固定長以外の仕様が似た**異なるTarget間で実際にベクトル化の性能差を評価1と同様に**コンパイル時のLV, SLPのStats比較**により探索
  - キャッシュ/レジスタ個数などを揃えた以下の２つのTargetを対象
    - x86-64 + AVX512
    - AArch64(ARMv8.5) + SVE



評価2: SVEとAVX512の自動ベクトル化の性能差発見

☀10,000回比較 (約8時間半👍)

	LoopVectorize Loops	SLPVectorizer Instructions
AVX512 wins	9	92
SVE wins	15	201

本研究提案のテスト生成①手法を適用後

	LoopVectorize Loops	SLPVectorizer Instructions
AVX512 wins	0	90
SVE wins	0	204

CSmithをそのまま利用

😄LoopVectorizeの結果は提案したテスト生成により発生

🙃SLPの結果は差分は見つかるが、テスト生成による変化はほぼなし

→**よりMutationも多様にすればLoopVectorizationの**
バグ/機会探索能力は期待できる

アウトライン

- 背景
  - ベクトル拡張命令/自動ベクトル化/差分テストを用いたコンパイラFuzzing
- 提案手法
  - Vectorizer Fuzzer
  - 評価
- 関連研究, まとめ/今後の展望

関連研究: Vectorizerアルゴリズム自体の改善

- All you need is superword-level parallelism: systematic control-flow vectorization with SLP [Chen+, PLDI '22]
  - LoopFusionやLoopUnrollngをSLPVectorizerに組み合わせた制御構造を多く扱えるアルゴリズム gSLPの提案
- Vectorization-aware loop unrolling with seed forwarding [Rocha+, CC '20]
  - 一般的でHeuristicな最適化であるLoopUnrollingのSLPVectorizationの有効活用を志向したHeuristicの提案
- Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks [Mendis+, ICML '19]
  - 機械学習を利用したハードウェアのコストモデル管理
- Compiler auto-vectorization with imitation learning [Mendis+, NeurIPS '19]
  - SLPVectorizerのコストモデルの計算ヒューリスティックに Graph Neural networkを用いる

まとめ・今後の課題

まとめ 🎯

☀ **LoopVectorizer**のTarget間の性能差発見に効果的な①**テスト生成**, ②**差分テスト**の提案

- Clang(2023/12 trunk), Arm SVE, x86 AVX512を対象に
  Target間のLoopVectorizer性能差発見

今後の課題

☀ 見つかった差分の解析部分の実装, バグやベクトル化機会の発見

- ISAレベルのモデル化 e.g. Lane Level Parallelism [Chen+, ASPLOS '21]
- 関連研究ではCReduceを使用 [John+, PLDI '12]

☀ コンパイラ非依存のComparatorによるコンパイラ間(gcc, etc)の比較

Appendix

LoopVectorization Legality Example

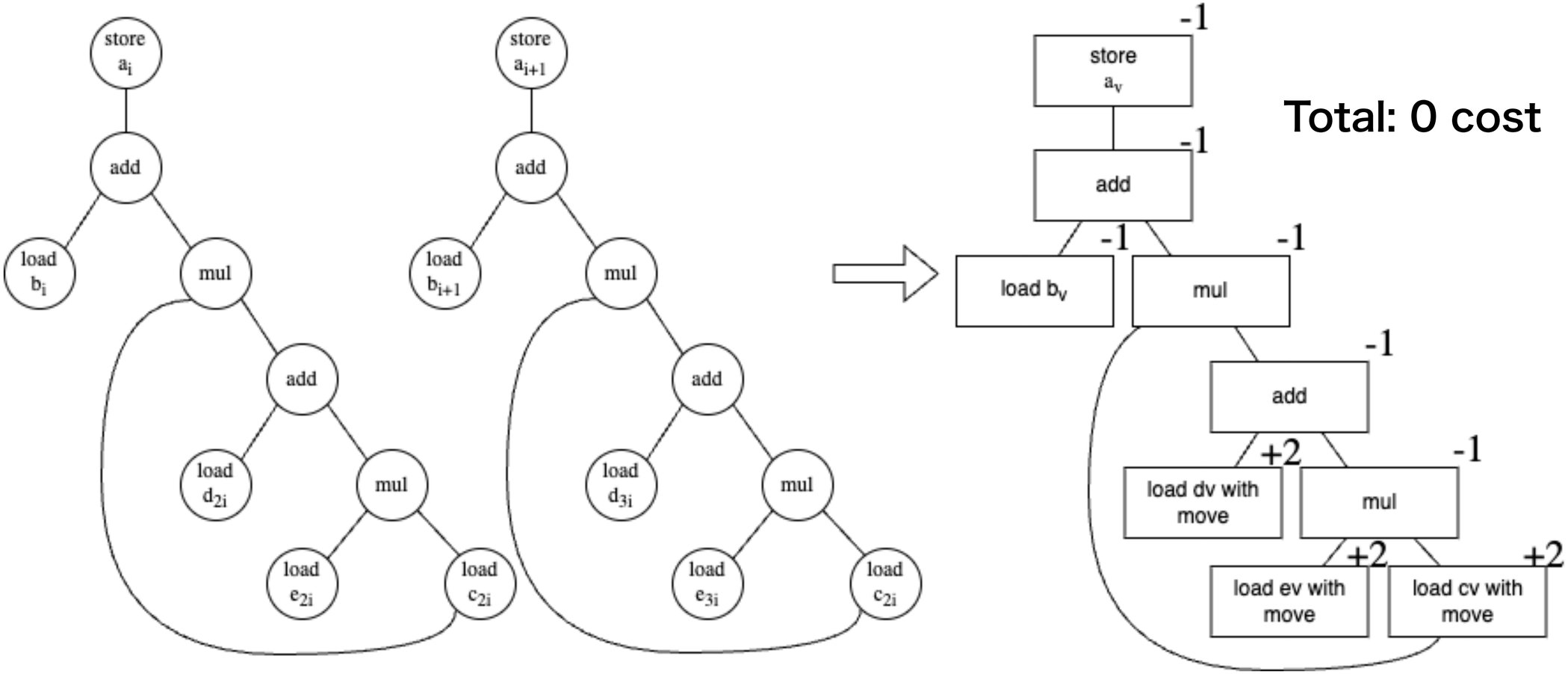
```
// ✗ ループ間にメモリの依存が入るのでInduction失敗
int memory_dep(int a[100]) {
    for (int i=0;i<100;++i) {
        int t = a[i];
        a[i] = a[i+1];
        a[i+1] = t;
    }
}
// ✗ メモリが可変なためReduction失敗
int volat(volatile int* t, int a[100]) {
    *t = 0;
    for (int i=0;i<100;++i) {
        *t += a[i];
    }
    return *t;
}
```

```
// △ aliasingしているとベクトル化できないので分岐
int may_alias(int a[100], int b[100]) {
    for (int i=0;i<100;++i) {
        int t = a[i];
        a[i] = b[i+1];
        b[i+1] = t;
    }
}
// ○ 分岐は入らず
int no_alias(int a[restrict 100],
             int b[restrict 100]) {
    for (int i=0;i<100;++i) {
        a[i] = 2 * b[i];
    }
}
```

https://godbolt.org/z/E77hvbdz6

SLPVectorization Profitability Example

$$a_i = b_i + c_{2i} * (d_{2i} + (e_{2i} * c_{2i}))$$
$$a_{i+1} = b_{i+1} + c3_i * (d_{3i} + (e_{3i} * c_{3i}))$$



Total: 0 cost

Reproduced from [Porpodas +, PACT 2015]

関連研究: 最適化の自動化 Superoptimizer

☀ Minotaur: A SIMD-Oriented Synthesizing Superoptimizer [Zhengyang+, arxiv '23]

☀ Faster sorting algorithms discovered using deep reinforcement learning [Zhengyang+, arxiv '23]

- 強化学習をアセンブリ命令の並び替えに適用し,
  実用的なライブラリの改善を実現

関連研究:差分テストベースのコンパイラFuzzing

- Finding missed optimizations through the lens of dead code elimination [Theodoros+, ASPLOS '22]
  - DeadCodeMarkerを利用し最適化機会を自動探索
- UBfuzz: Finding Bugs in Sanitizer Implementations [Shaohua+, ASPLOS '24]
  - Debug情報を利用しSanitizerの偽陽性を自動探索
- Compilation Consistency Modulo Debug Information [Theodore+, ASPLOS '23]
  - デバッグコンパイル(-g)の不整合を自動探索

先行研究のframeworkとの対応

	Dead [Theodoros+,]	Ubfuzz [Shaohua+,]	Dfuser [Theodore+,]	VecFuzz
Generator	CSmith	CSmith	CSmith	Smith
Mutator	各BasicBlockに マクロを挿入	演算, 配列アクセスなどの 箇所への未定義動作挿入	1) コンパイラディレクティブの挿入 2) 制御構造の変更	1) ループを4048 trip 2) 配列を128 bytes 3) 配列を4048 サイズ
Compile opts	1) -O2 vs -O3, 2) GCC vs LLVM 3) Clang/GCC version A vs B	-O0 vs -O3	-O3 -g vs -O3	次項参照
Compare	挿入したマクロの 在否を比較	“-O0の Crash => -O3のCrash” を検証	各バイナリ中の 命令を比較	SLPVectorizer, LoopVectorizeの ベクトル化回数を利用

評価2: 実験のコンパイルオプションの詳細

```
~/git/llvm-project/build/bin/clang /home/khei4/git/VecFuzz/veccmptest0.c -S -o tmp.s \
    -isystem/home/khei4/git/VecFuzz/csmith/include \
    -isystem/usr/aarch64-linux-gnu/include/ -std=c23 -O3 -target \
    aarch64 -march=armv8.5-a+sve -msve-vector-bits=512 -mllvm -stats \
    -emit-llvm -Wno-everything -mllvm -slp-threshold=-100 2> aarch64_tti.txt
~/git/llvm-project/build/bin/clang /home/khei4/git/VecFuzz/preproced2x86.c -S -o tmp.s \
    -isystem/home/khei4/git/VecFuzz/csmith/include -std=c23 \
    -O3 -target x86_64-pc-linux-gnu -mavx512f -mllvm -stats \
    -emit-llvm -Wno-everything -mllvm -slp-threshold=-100 2> x86_tti.txt
```

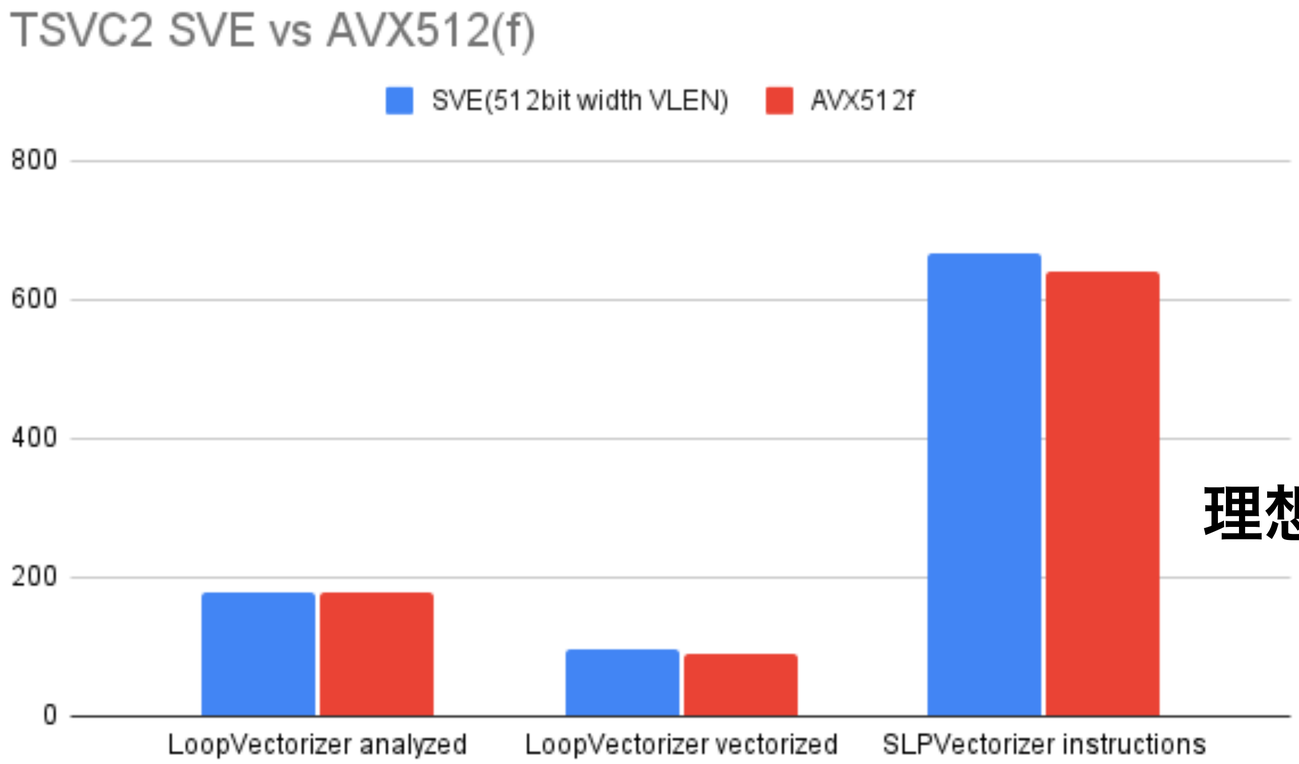
Table 4.1: SVE and X86 TargetTransformInfo on Evaluation

Target	Fixed Vector Register Width	Scalable Vector Register Width	MinVector Register Width
x86-64 AVX512	512	0	128
AArch64 SVE	512	128	64

1) ISA以外の観点はArm SVEを優位に, 2) SLPのコストモデルを無視, した

評価2: TSVC2 に対するx86-64+avx512 とARMv8+sve(512bw) のStats予備比較

☀ AutoVectorization向けのTSVC benchmark[Callahan+, 1988]を用いて
前項のコンパイル設定での性能を比較



理想的なベンチマークでは
Arm SVEが優位

*実験にはTSVC2 を利用 https://github.com/UoB-HPC/TSVC_2

解析(Future work)

1. Generator
2. Mutator

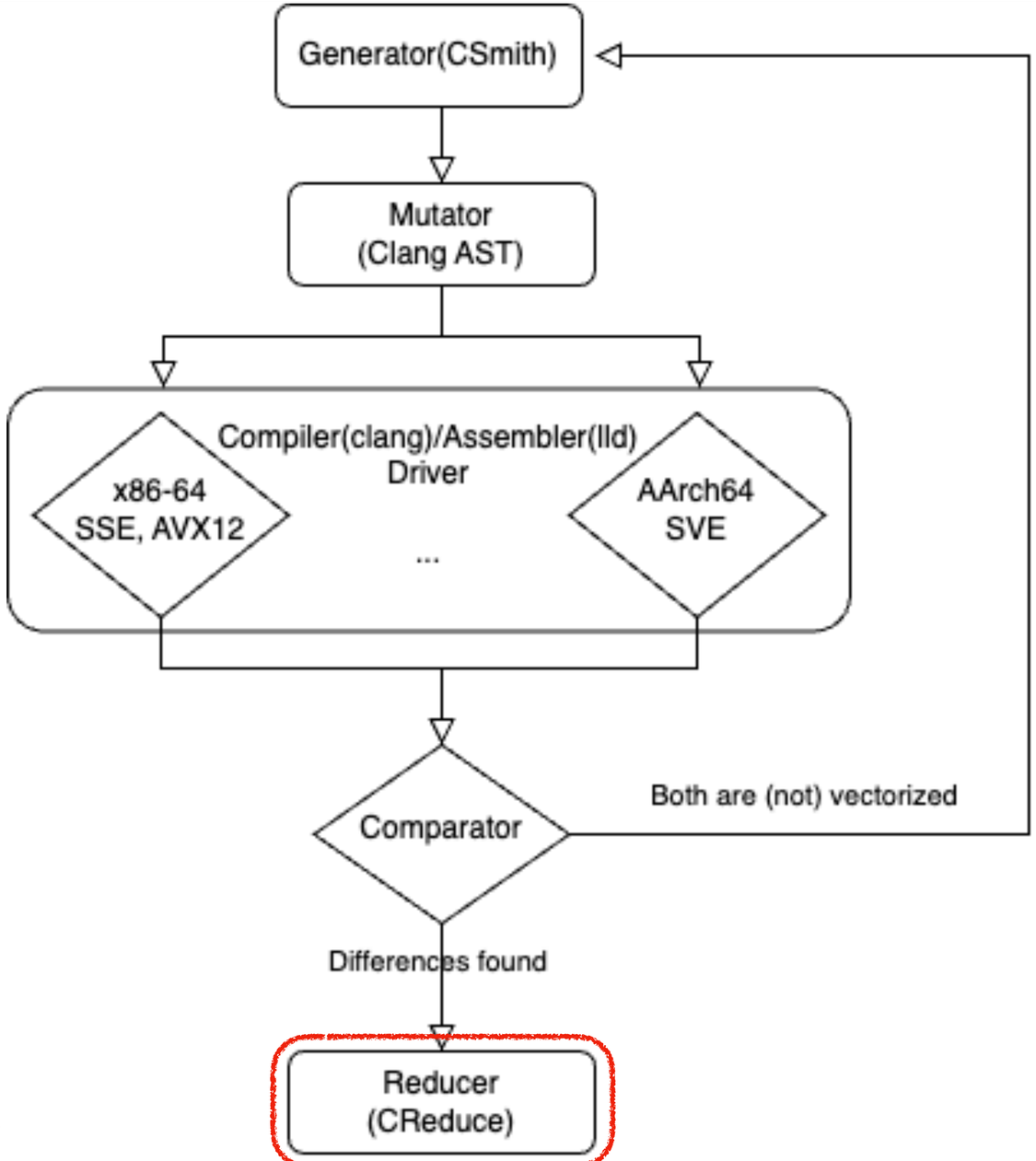
テスト生成

3. Compiler(s)
4. Comparator

差分テスト

5. Reducer

解析



Future work: Reducer Example (for out of topic issue)

1. Generator
2. Mutator
3. Compiler(s)
4. Comparator
5. **Reducer**

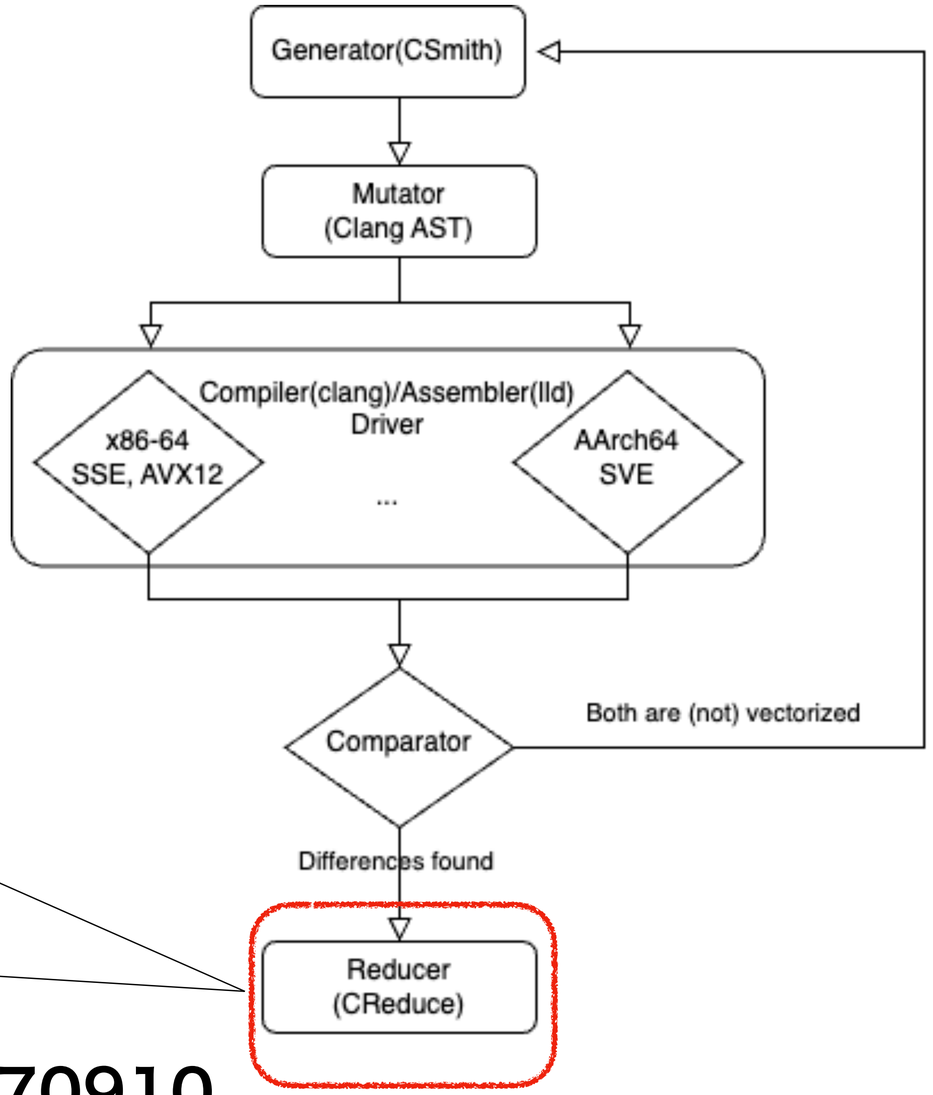
Generated C program

```
# 1 "./csmith_tmp.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 396 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "./csmith_tmp.c" 2
# 10 "./csmith_tmp.c"
# 1 "/home/khei4/git/VecFuzz/csmith/
include/csmith.h" 1
# 40 "/home/khei4/git/VecFuzz/csmith/
include/csmith.h"
# 1 "/usr/include/string.h" 1 3 4
# 26 "/usr/include/string.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/
libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/
libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 439 "/usr/include/features.h" 3 4
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 440 "/usr/include/features.h" 2 3 4
# 461 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/
cdefs.h" 1 3 4
# 452 "/usr/include/x86_64-linux-gnu/sys/
cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/
wordsize.h" 1 3 4
# 453 "/usr/include/x86_64-linux-gnu/sys/
cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/
long-double.h" 1 3 4
# 454 "/usr/include/x86_64-linux-gnu/sys/
cdefs.h" 2 3 4
# 462 "/usr/include/features.h" 2 3 4
# 485 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/
stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/
stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/
stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/
stubs.h" 2 3 4
# 486 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/
libc-header-start.h" 2 3 4
# 27 "/usr/include/string.h" 2 3 4
# 1 "/home/khei4/git/llvm-project/build/
lib/clang/18/include/stddef.h" 1 3 4
# 77 "/home/khei4/git/llvm-project/build/
lib/clang/18/include/stddef.h" 3 4
# 1 "/home/khei4/git/llvm-project/build/
lib/clang/18/include/__stddef_size_t.h" 1 3
4
# 13 "/home/khei4/git/llvm-project/build/
lib/clang/18/include/__stddef_size_t.h" 3 4
typedef long unsigned int size_t;
# 78 "/home/khei4/git/llvm-project/build/
lib/clang/18/include/stddef.h" 2 3 4
# 92 "/home/khei4/git/llvm-project/build/
lib/clang/18/include/stddef.h" 3 4
# 1 "/home/khei4/git/llvm-project/build/
```

$4.5k$ LoC

Reduced

```
int a[3183476736] = {5};
```



https://godbolt.org/z/3ddzP8P3b

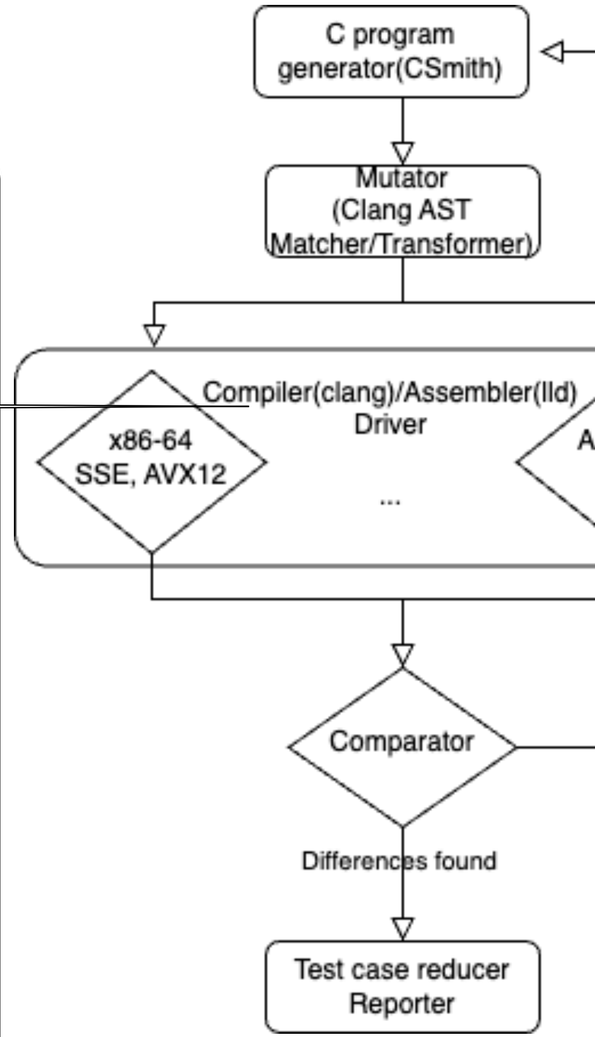
https://github.com/llvm/llvm-project/issues/70910

Mutator(Internal): (ClangAST Transformer)

1. Generator
2. **Mutator**
3. Compiler(s)
4. Comparator
5. Reducer

```
…
  return makeRule(
      makeNDimensionalArrayMatcher(n),
      flatten(ifBound(
          "arrayInitializer",
          changeTo(cat("alignas(128) ",

between(before(node("nDimArray")), name("nDimArray")),
                       name("nDimArray"), NewSize,
"=",
                       node("arrayInitializer"),
";")),
          changeTo(cat(
              "alignas(128) ",
              between(before(node("nDimArray")),
name("nDimArray")),
              name("nDimArray"), NewSize, ";"
               )))));
```



予備:評価2 詳細な数字数字

```
[04:12] koheis-MacBook-Air.local:pa
mode =  SLP
before vecmut total diff num:  294
before vecmut num sve wins:  204
before vecmut num avx512 wins:  90
after vecmut total diff num:  293
after vecmut num sve wins:  201
after vecmut num avx512 wins:  92
diff lost by vecmut:  78
diff found only if vecmut:  77
```

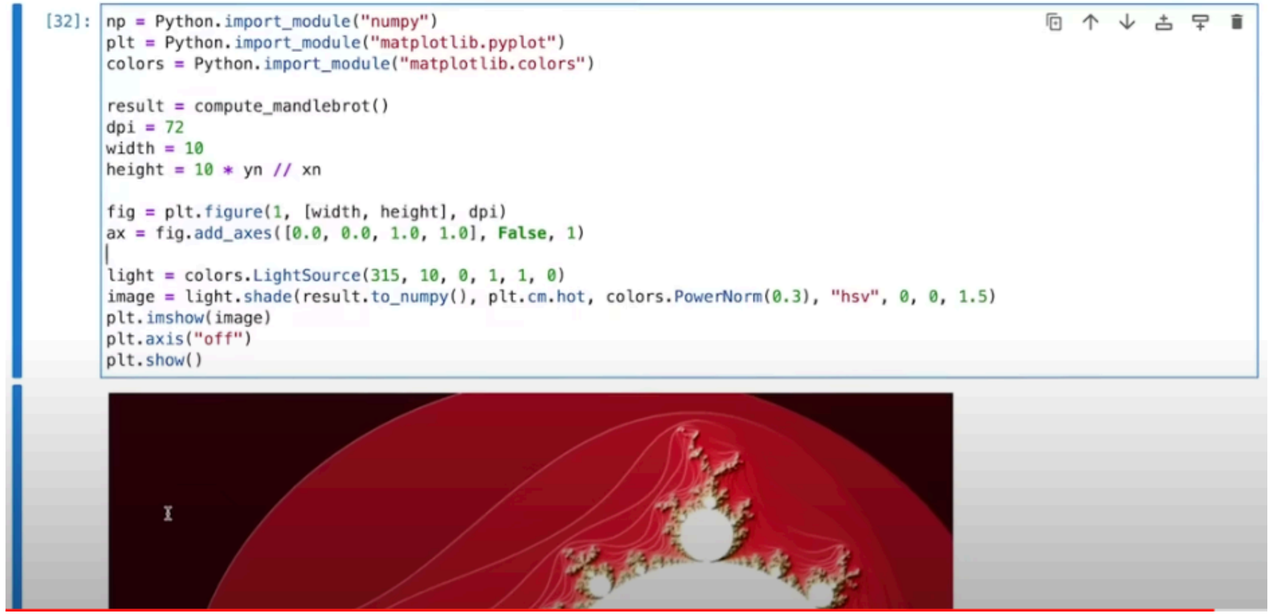
```
[04:12] koheis-MacBook-Air.local:p
mode =  LV Loop Num
before vecmut total diff num:  0
before vecmut num sve wins:  0
before vecmut num avx512 wins:  0
after vecmut total diff num:  24
after vecmut num sve wins:  15
after vecmut num avx512 wins:  9
diff lost by vecmut:  0
diff found only if vecmut:  24
```

diff lost by Mutator … Mutator を適用しないCSmithの生のCでのみ見つかった差

diff found by Mutator … Mutator を適用後のCでのみ見つかった差

予備:ベクトル拡張命令のポテンシャル

☀ Mojo🔥はMLIRを用いたSIMD命令の有効活用で特定のベンチマークに対して Pythonに対する35000倍の高速化を達成した



* https://www.youtube.com/watch?v=6GvB5lZJqcE

予備: SLPVectorizerで差がある結果を手でReduce

```
...
static uint64_t func_13(uint8_t *p_14, uint8_t *p_15,
                        const int16_t p_16) { /* block id: 6 */

  uint16_t *l_50 = &g_51[3];
  int64_t *l_2127 = &g_2128;
  uint32_t p_55 = 0;
  for (g_106 = 0; (g_106 <= 3); g_106 += 1) {
    for (g_202 = 1; (g_202 <= 5); g_202 += 1) {
      for (g_26 = 0; (g_26 <= 3); g_26 += 1) {
        int32_t *l_1280 = &g_1281;
        alignas(128) int32_t l_1944[4096];
        int32_t *l_2076 = &l_1944[1];
        int32_t *l_2077 = &g_1567;
        int64_t *l_1939 = &g_151[0];
        alignas(128) int32_t l_2111[4096];
        alignas(128) uint16_t l_2074[4096][4096][4096] = {
            {{5UL, 0xD149L, 4UL, 4UL, 0xD149L}},
            {{0xDAC9L, 0x0BB9L, 4UL, 0x5CE6L, 0x5CE6L}},
            {{0x0BB9L, 0xDAC9L, 0x0BB9L, 4UL, 0x5CE6L}}};
        uint16_t *volatile ***l_2071 = &g_2070[1];
        uint16_t ***l_2093 = (void *)0;
        (*l_1280) =
            ((*l_2077) = (safe_mul_func_uint64_t_u_u(
                 p_55,
                 ((*l_1939) |=
                  ((safe_mul_func_int16_t_s_s(
                       (((safe_lshift_func_int32_t_s_s(
                            ((*l_2076) = (safe_mul_func_int16_t_s_s((*g_1243),
                                                                    65531UL))),
                            25)) ^
                          (l_2093 == (*l_2071))) |
                         0L),
                       (safe_rshift_func_uint32_t_u_u(
                           ((safe_mod_func_uint8_t_u_u(
                                (((*g_1244) = (*g_1244)) ==
                                 (((safe_div_func_int8_t_s_s(
                                       (safe_lshift_func_uint16_t_u_u(0, 0)),
                                       (-10L))),
                                   l_2111[4]),
                                  &g_239)),
                                p_55)) != 0),
                            l_2074[1][0][3]))))) ^
                   p_55)))));
      }
    }
  }

{ /* block id: 1020 */
  //   TODO: -> 13, 9
  for (g_500 = 0; (g_500 <= 8); g_500 += 1) {        /* block id: 1160 */
    for (g_2248 = 3; (g_2248 <= 8); g_2248 += 1) {   /* block id: 1164 */
      for (l_1281 = 0; (l_1281 >= 0); l_1281 -= 1) { /* block id: 1167 */
        alignas(128) int32_t l_2320[4096];
        int32_t l_2625 = 1L;
        int32_t l_2626 = (-9L);
        int8_t *l_2658 = &g_2646;
        uint16_t **l_2630 = &l_50;
        //   TODO: -> 13, 9
        l_2320[0] =
            ((safe_sub_func_uint16_t_u_u(
                 (++(**l_2630)),
                 ((0UL ^ g_1158[g_2248]) ==
                  ((*l_2658) = (safe_lshift_func_int32_t_s_s(
                       (safe_rshift_func_int32_t_s_s(
                           (safe_rshift_func_int32_t_s_s(
                               ((*g_103) &
                                (l_2626 ^= (safe_mod_func_uint8_t_u_u(
                                     ((*p_14) !=
                                      ((safe_rshift_func_int16_t_s_u(
                                           g_2646, (safe_add_func_int16_t_s_s(
                                                p_16, 0x253CL)))) >
                                       false)),
                                     l_2625)))),
                               14)),
                           p_16)),
                       14)))))) == 0xDEA0201BL);
      }
    }
  }
  return NULL;
}
```

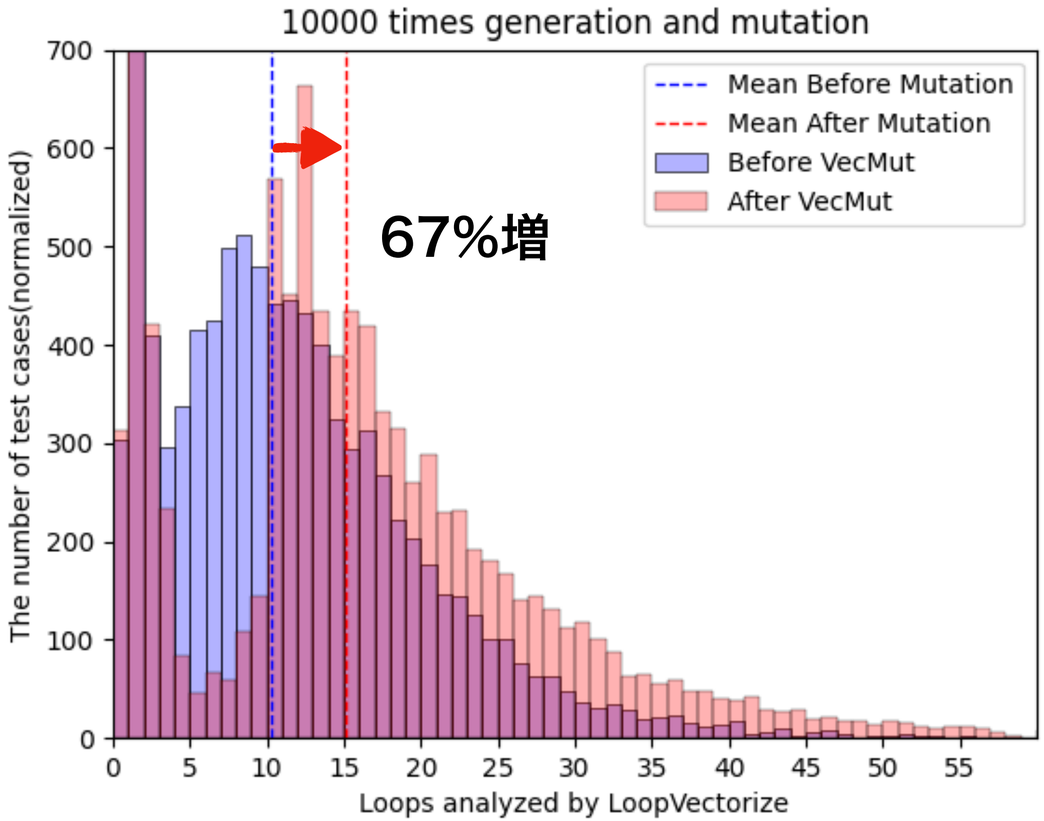
LVの差があるケース https://godbolt.org/z/rfr6eveo9, SLPの差があるケース https://godbolt.org/z/j6xnMceGx

評価1において, LoopVectorizeがLegalityの検査を行ったLoop数

☀ 10,000回の試行のうちLoopVectorizeが**解析するループ**の個数

- 1ケースあたりループ個数は**67%増**
  - **7142**ケースで増加(724ケース減少)

Epilogueループの生成などに連鎖して,
実際のVectorized loopより増えていると推測



LLVMのコンパイル時最適化Statsの例

☀ LLVMをAssertion buildし, Clangに -mllvm stats コマンドを渡すと下記のような統計値が見れる

```
4 SLP - Number of vector instructions generated
4 loop-vectorize - Number of loops analyzed for
vectorization
1 loop-vectorize - Number of loops vectorized
```

- SLPが**ベクトル化した命令**数
- LoopVectorizeが**解析したループ**数
- LoopVectorizeが**ベクトル化したループ**の数