Linaro Connect

SVE and SVE2 in LLVM

Will Lovett – Principal Technology Manager

Development Solutions Group, Arm

will.lovett@arm.com

@hpc_will

25th March 2021

Agenda

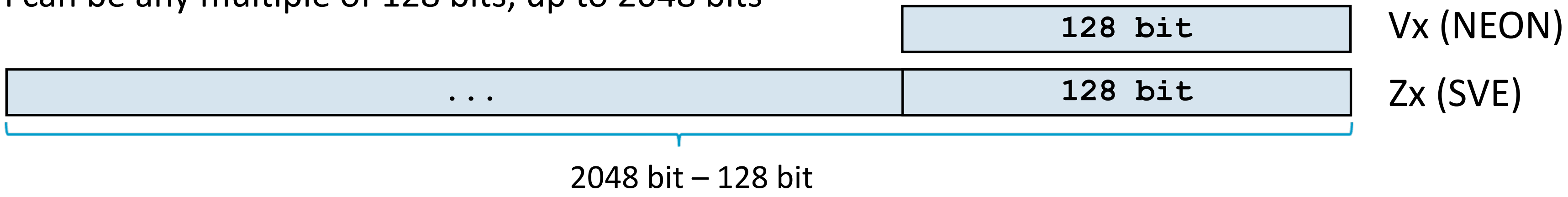
Follow along with the examples: **tinyurl.com/LVC21-SVE**

- Introduction
- Status of SVE in LLVM
- Code examples
- Roadmap

Introduction to SVE and further reading

SVE/SVE2

- Scalable vector extension to the Arm v8-a architecture
- Vector length can be any multiple of 128 bits, up to 2048 bits



- Predicate registers allow conditional execution of individual lanes within the vector
- Find out more: [SVE & SVE2 Programmer’s Guide]

ACLE

- Arm C Language Extensions
- C intrinsics that map (roughly) 1-1 with Arm instructions
- ACLE for SVE covers support for SVE, SVE2 and Arm v8.6-a extensions to SVE
- Find out more: [Arm C Language Extensions Specification]

Status today

LLVM 9 (September 2019)

- SVE/SVE2 assembly and disassembly

LLVM 11 (September 2020)

- SVE/SVE2 intrinsic (ACLE) support
- Armv8.6-a support (bfloat16, matmul)

LLVM 12 (March 2021)

- ACLE Stabilisation and code quality improvements
  - Support for vector-length specific ACLE
  - Debugger support for ACLE types
- Vector-length specific SVE autovectorization (functional, with performance issues)
- **Vector-length agnostic SVE vectorization of a few simple loops**

Status of SVE auto-vectorization

What does a production compiler need?

	Goals for LLVM 12	LLVM 13	LLVM 14
Safety	• Source changes with pragma		
Capability	• > 1 loop in TSVC		
Quality	• Terrible!		

Code examples

Today, we'll look at:

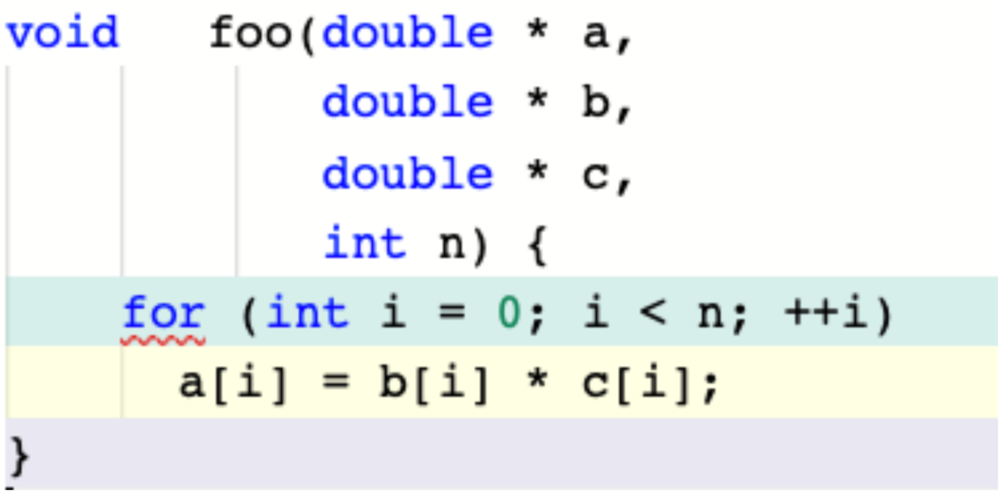
- Neon vectorization
- Simplify
- Enable SVE
- Non-contiguous data
- Use constants
- Invariant loads
- Conditional execution
- Reductions
- Use induction variable

Other topics we're ignoring:

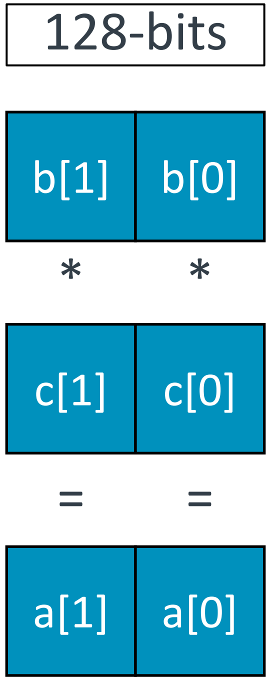
- Use of the ACLE
- Multiple exits and unknown trip counts
- Complex number support
- Cost modelling (to decide when to use SVE)
- Scalar tail removal
- SVE2-specific features
- Code quality
- … many more

Example 1: NEON vectorization

https://godbolt.org/z/enzqz5



```
void    foo(double * a,
            double * b,
            double * c,
            int n) {
    for (int i = 0; i < n; ++i)
      a[i] = b[i] * c[i];
}
```



128-bits

b[1] b[0]

* *

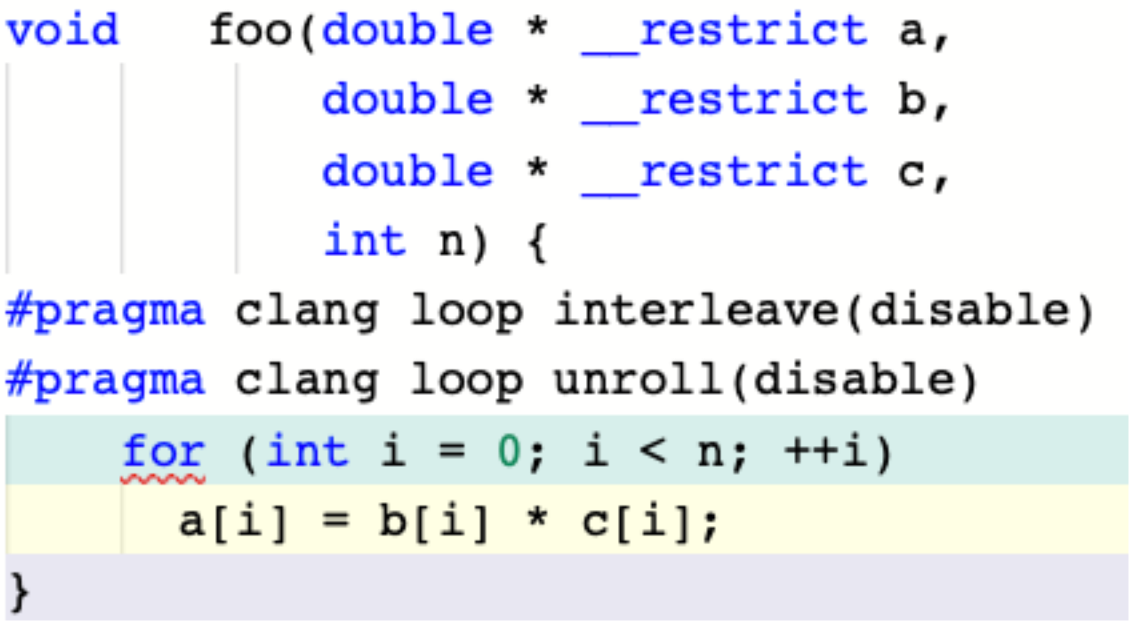
c[1] c[0]

= =

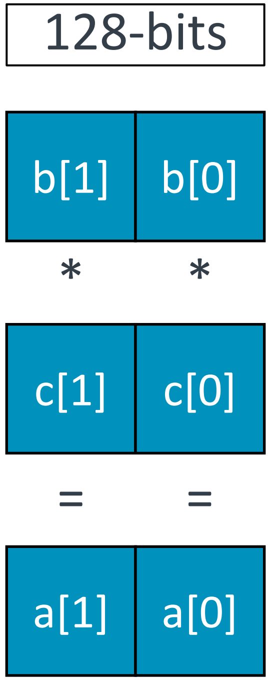
a[1] a[0]

Example 2: Simplify the output

https://godbolt.org/z/abf3sY

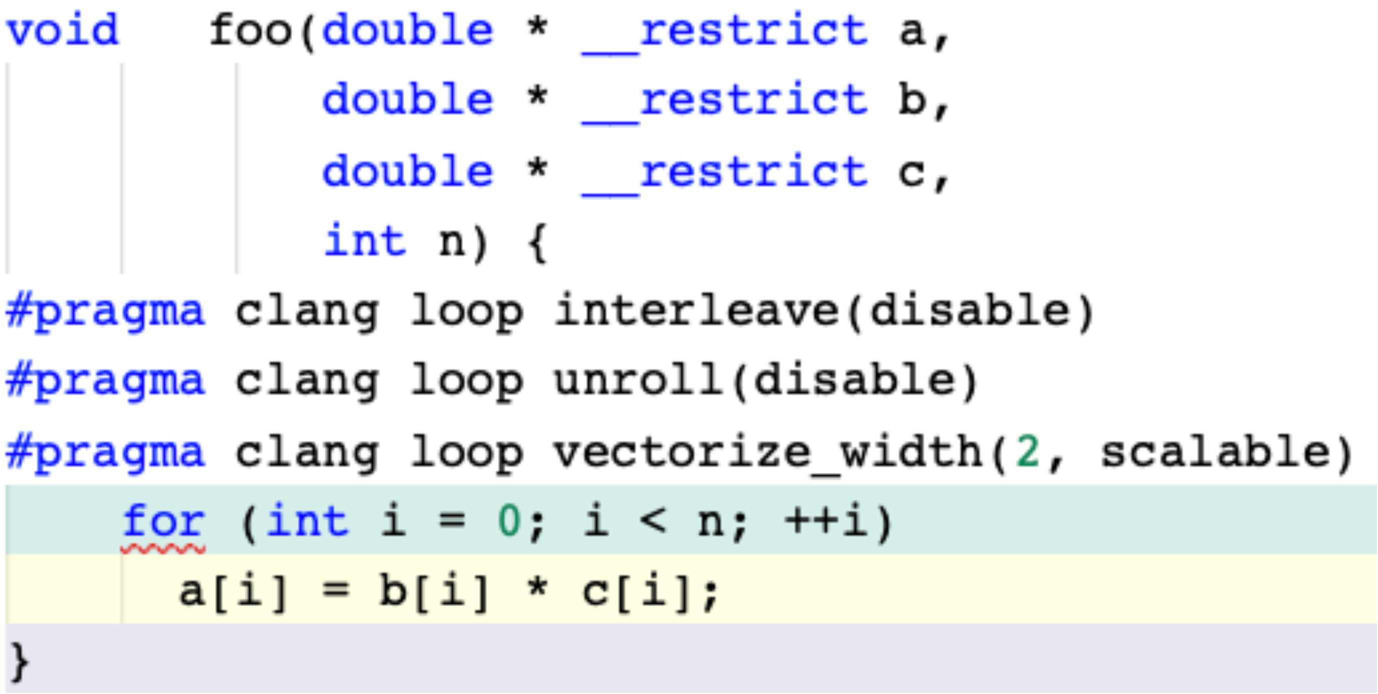


```
void    foo(double * __restrict a,
            double * __restrict b,
            double * __restrict c,
            int n) {
#pragma clang loop interleave(disable)
#pragma clang loop unroll(disable)
    for (int i = 0; i < n; ++i)
      a[i] = b[i] * c[i];
}
```

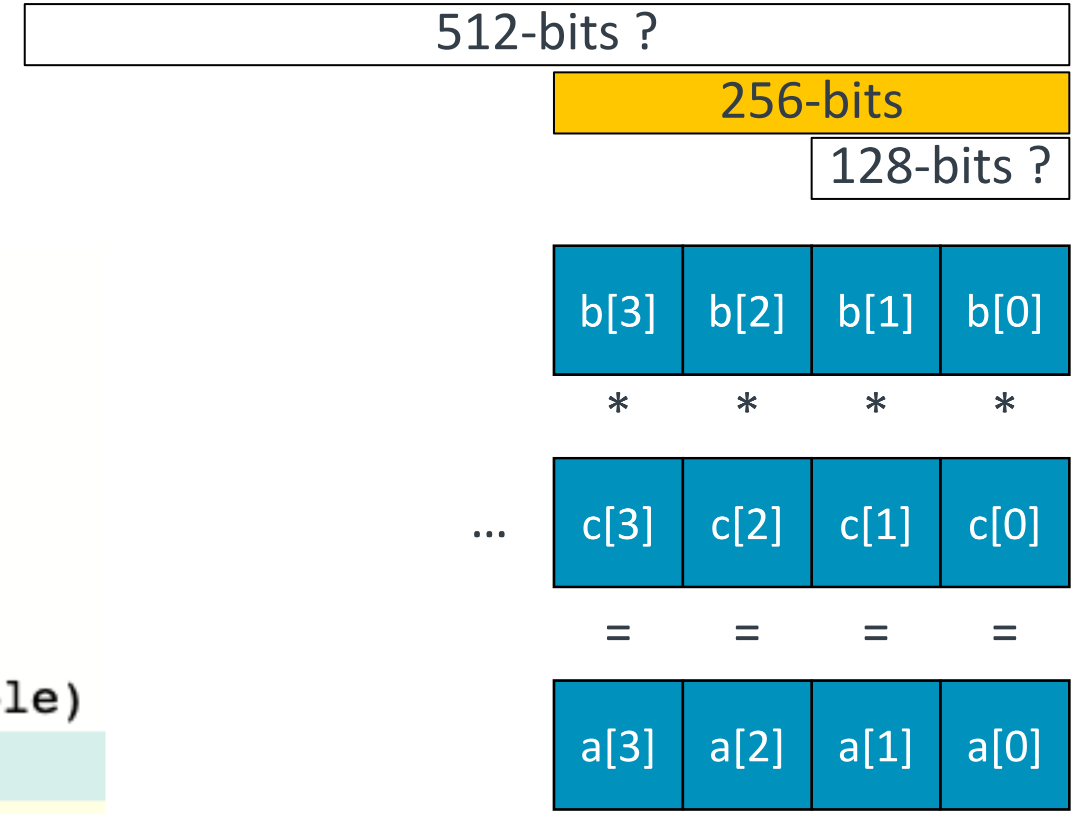


Example 3: Enable SVE

https://godbolt.org/z/a8W6z9

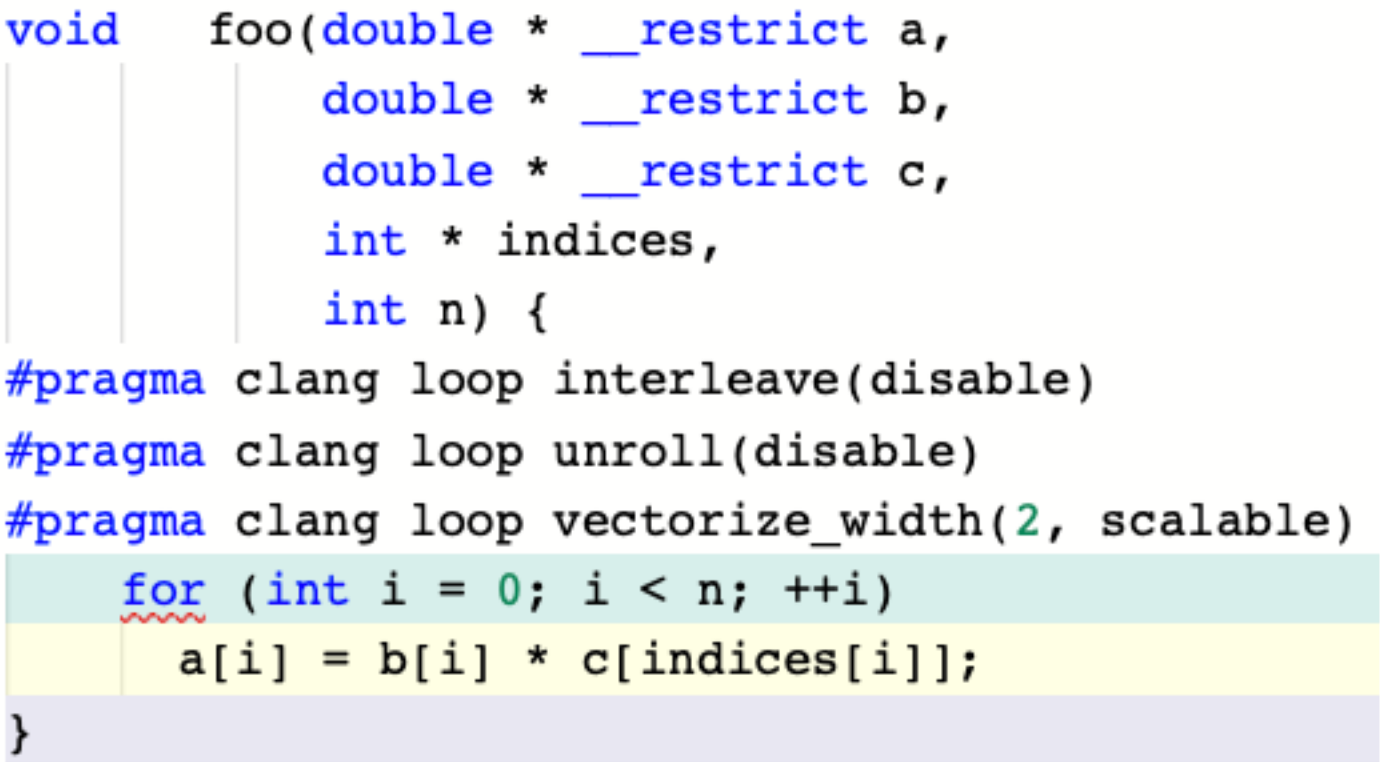


```
void    foo(double * __restrict a,
            double * __restrict b,
            double * __restrict c,
            int n) {
#pragma clang loop interleave(disable)
#pragma clang loop unroll(disable)
#pragma clang loop vectorize_width(2, scalable)
    for (int i = 0; i < n; ++i)
      a[i] = b[i] * c[i];
}
```

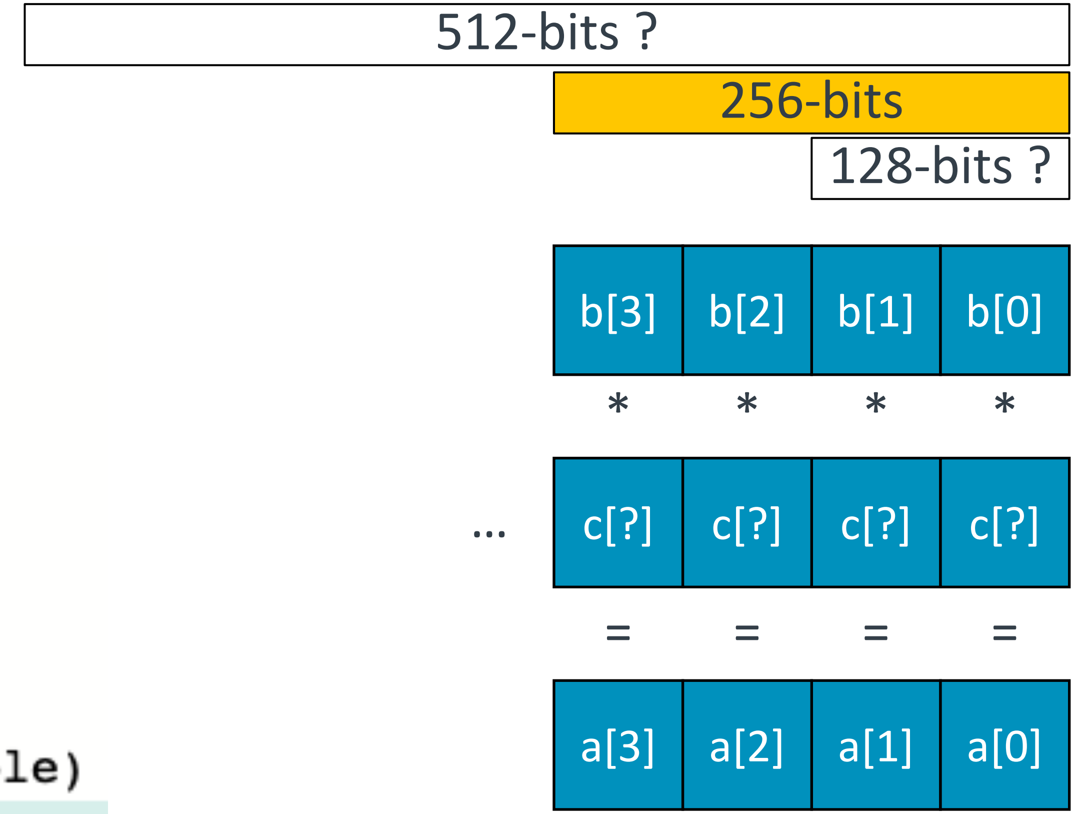


Example 4: Non-contiguous data

https://godbolt.org/z/1KKjoT



```
void    foo(double * __restrict a,
            double * __restrict b,
            double * __restrict c,
            int * indices,
            int n) {
#pragma clang loop interleave(disable)
#pragma clang loop unroll(disable)
#pragma clang loop vectorize_width(2, scalable)
    for (int i = 0; i < n; ++i)
      a[i] = b[i] * c[indices[i]];
}
```



512-bits ?

256-bits

128-bits ?

b[3] b[2] b[1] b[0]

* * * *

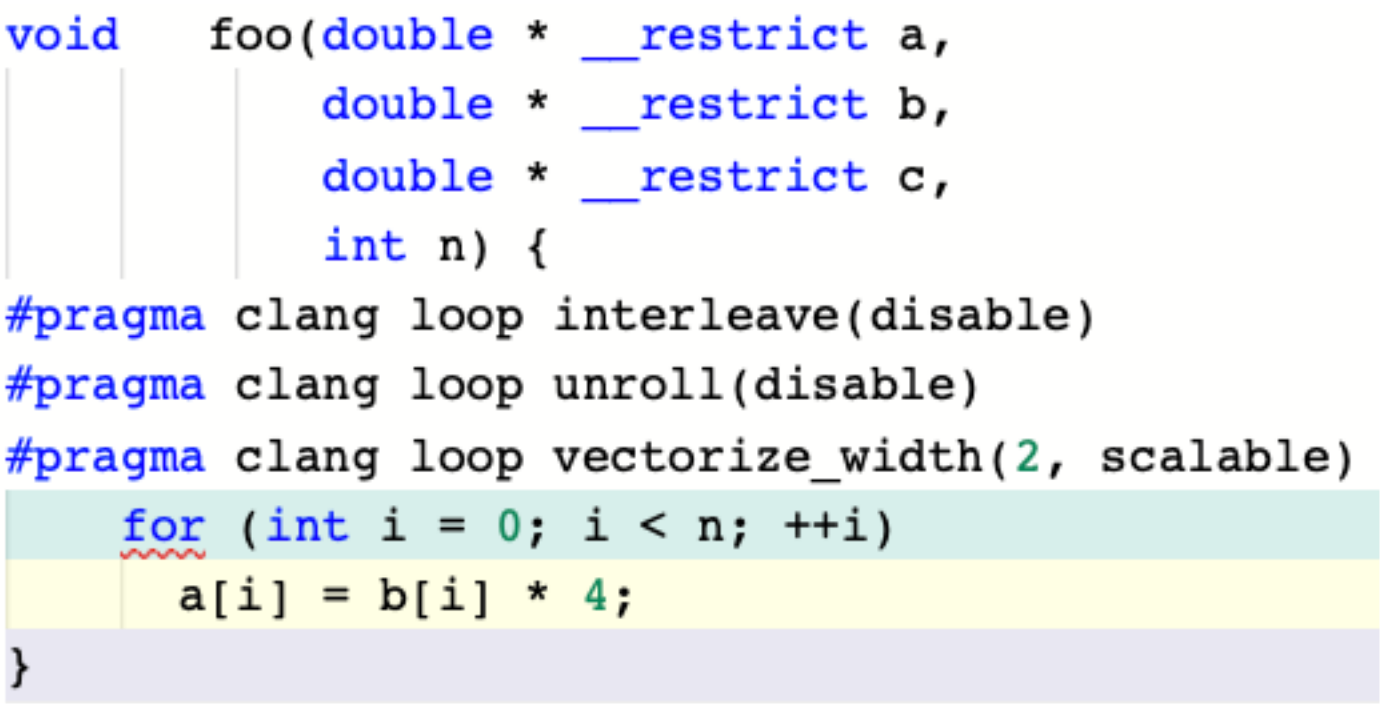
… c[?] c[?] c[?] c[?]

= = = =

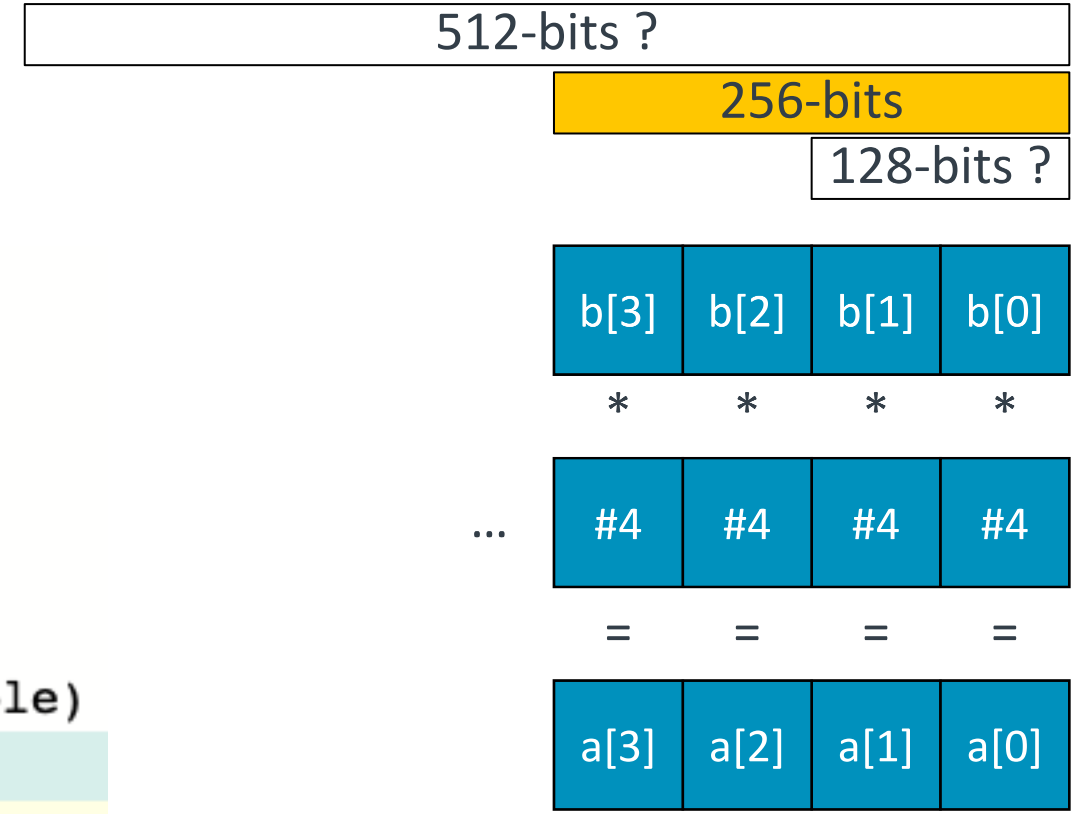
a[3] a[2] a[1] a[0]

Example 5: Use constants

https://godbolt.org/z/8YznWY

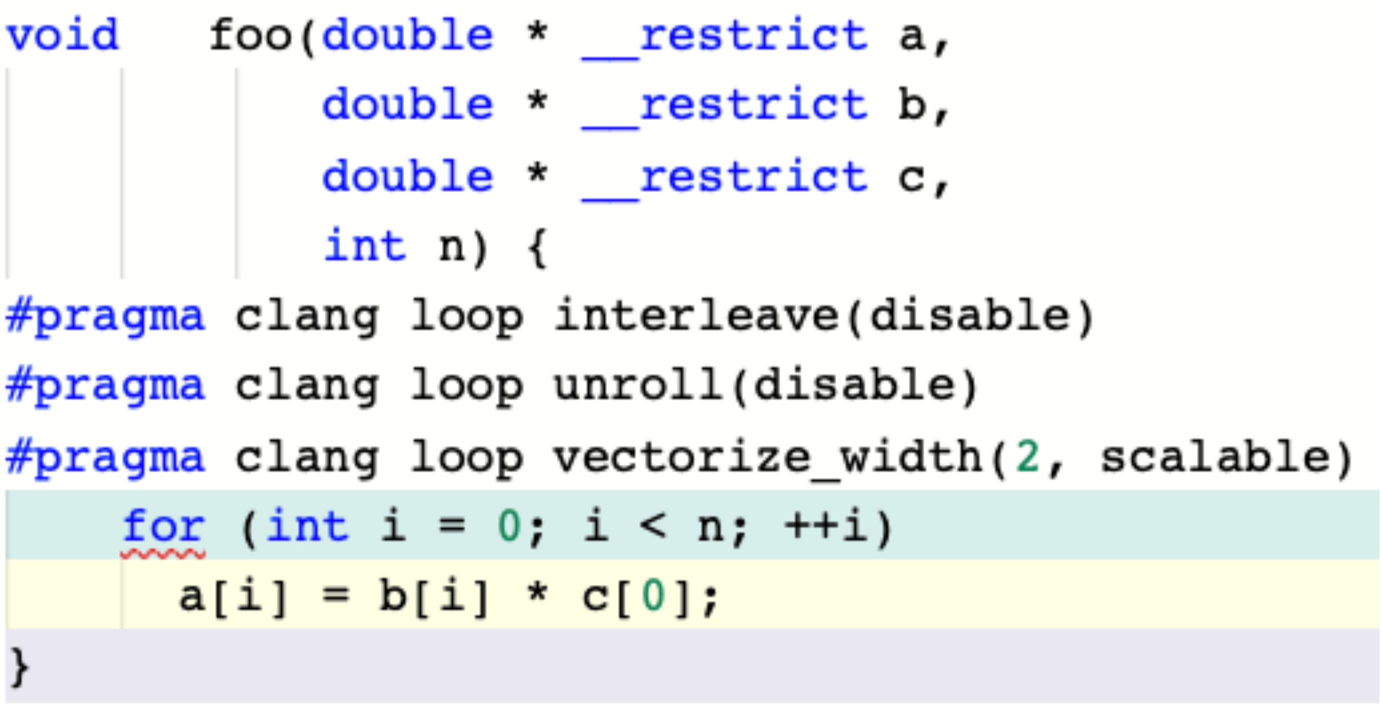


```
void    foo(double * __restrict a,
            double * __restrict b,
            double * __restrict c,
            int n) {
#pragma clang loop interleave(disable)
#pragma clang loop unroll(disable)
#pragma clang loop vectorize_width(2, scalable)
    for (int i = 0; i < n; ++i)
      a[i] = b[i] * 4;
}
```

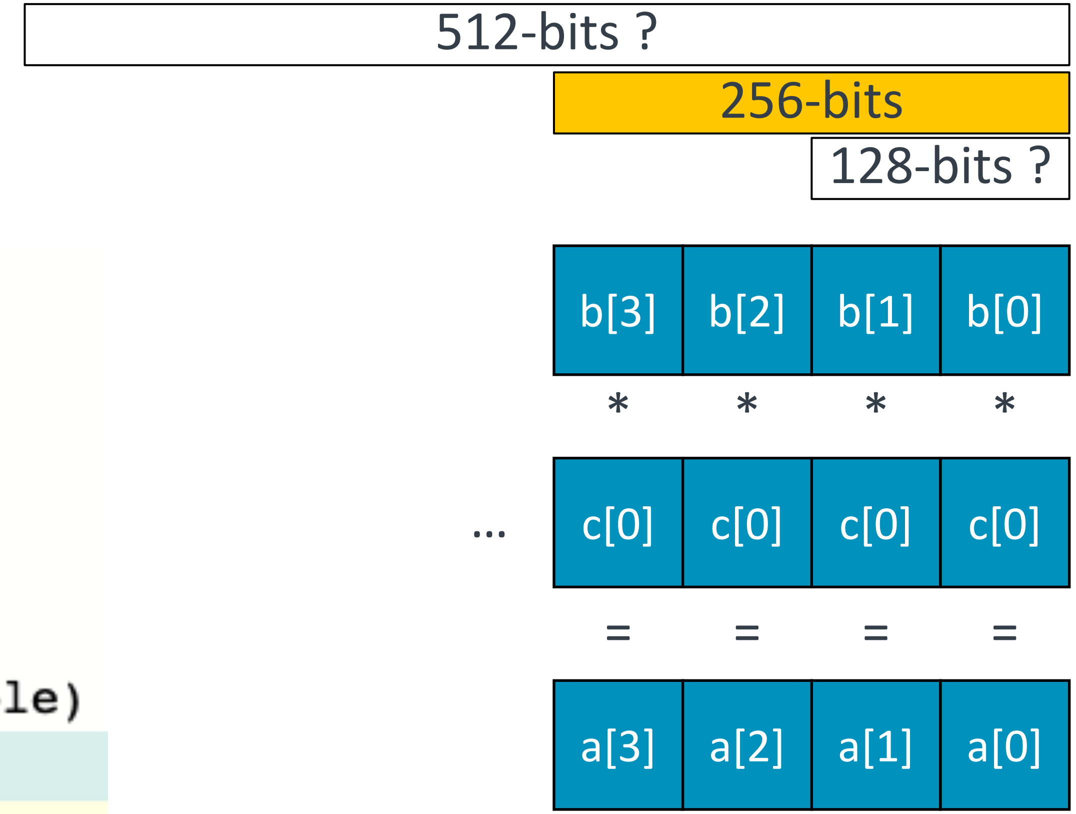


Example 6: Invariant loads

https://godbolt.org/z/8YKozv

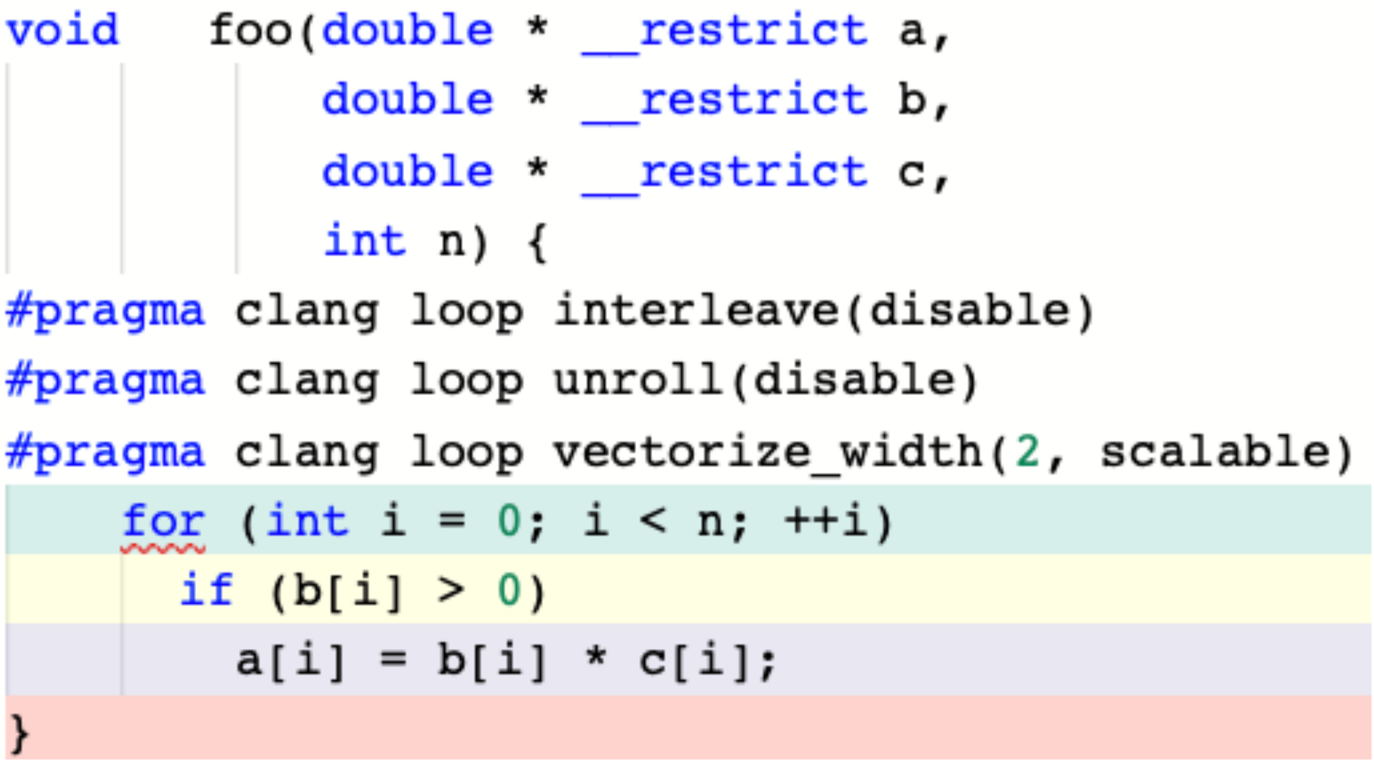


```
void    foo(double * __restrict a,
            double * __restrict b,
            double * __restrict c,
            int n) {
#pragma clang loop interleave(disable)
#pragma clang loop unroll(disable)
#pragma clang loop vectorize_width(2, scalable)
    for (int i = 0; i < n; ++i)
      a[i] = b[i] * c[0];
}
```

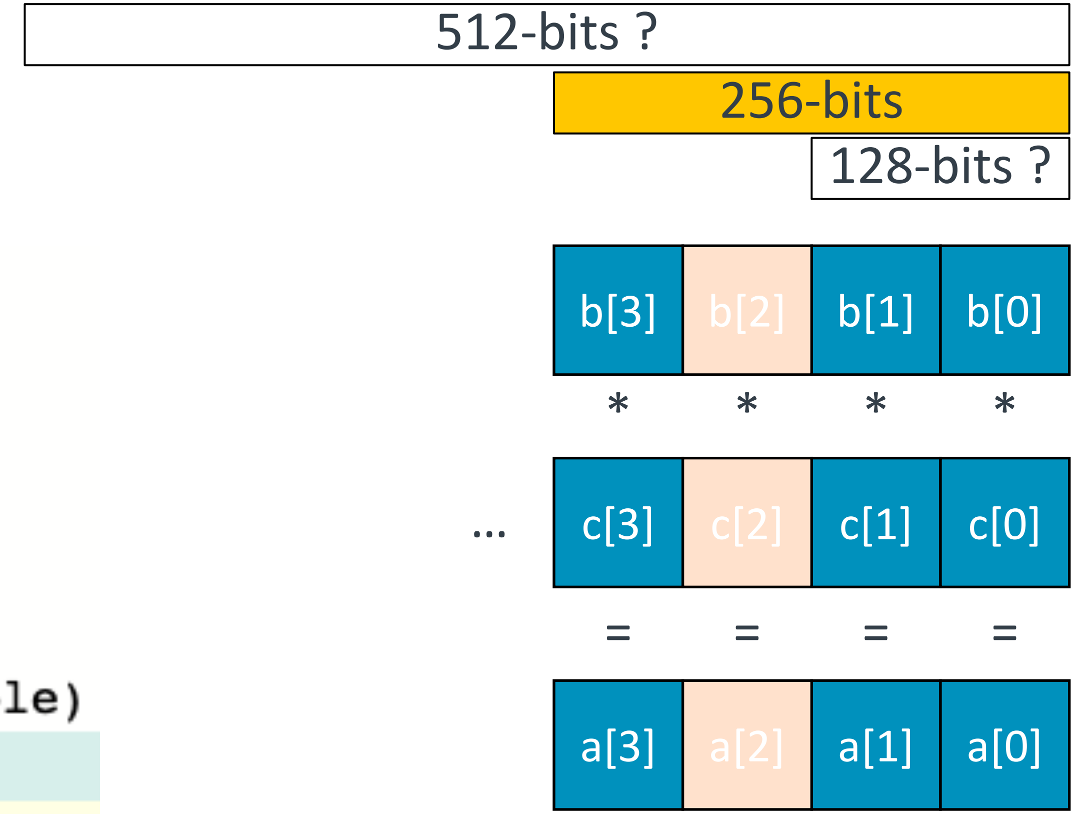


Example 7: Conditional execution

https://godbolt.org/z/bMYjMs



```
void    foo(double * __restrict a,
            double * __restrict b,
            double * __restrict c,
            int n) {
#pragma clang loop interleave(disable)
#pragma clang loop unroll(disable)
#pragma clang loop vectorize_width(2, scalable)
    for (int i = 0; i < n; ++i)
      if (b[i] > 0)
        a[i] = b[i] * c[i];
}
```



512-bits ?

256-bits

128-bits ?

b[3] b[2] b[1] b[0]

* * * *

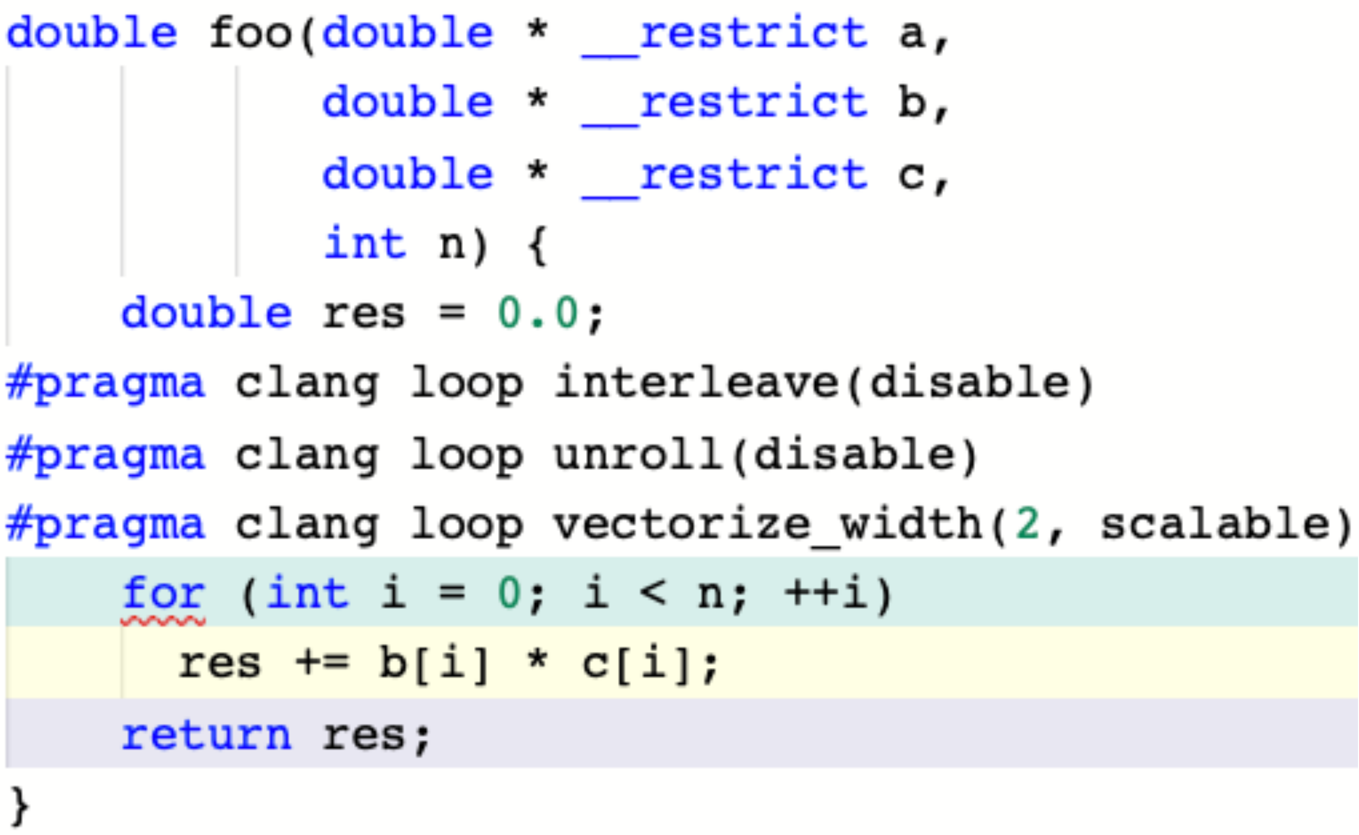
… c[3] c[2] c[1] c[0]

= = = =

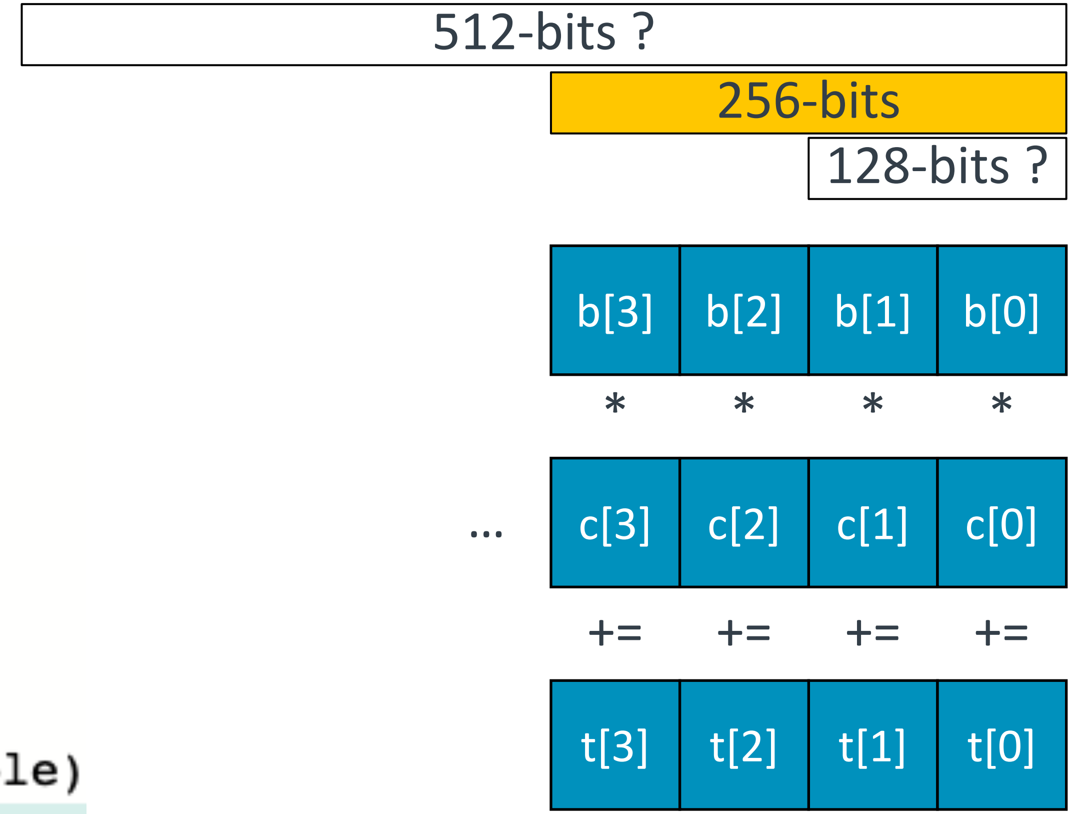
a[3] a[2] a[1] a[0]

Example 8: Reductions

https://godbolt.org/z/GzPMMP



```
double foo(double * __restrict a,
           double * __restrict b,
           double * __restrict c,
           int n) {
    double res = 0.0;
#pragma clang loop interleave(disable)
#pragma clang loop unroll(disable)
#pragma clang loop vectorize_width(2, scalable)
    for (int i = 0; i < n; ++i)
      res += b[i] * c[i];
    return res;
}
```



512-bits ?

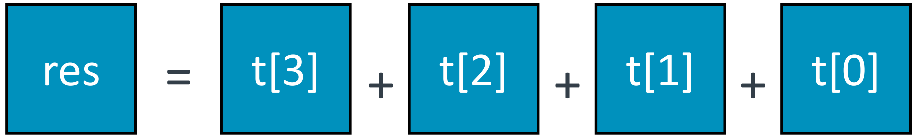
256-bits

128-bits ?

b[3]	b[2]	b[1]	b[0]
*	*	*	*
c[3]	c[2]	c[1]	c[0]
+=	+=	+=	+=
t[3]	t[2]	t[1]	t[0]

…

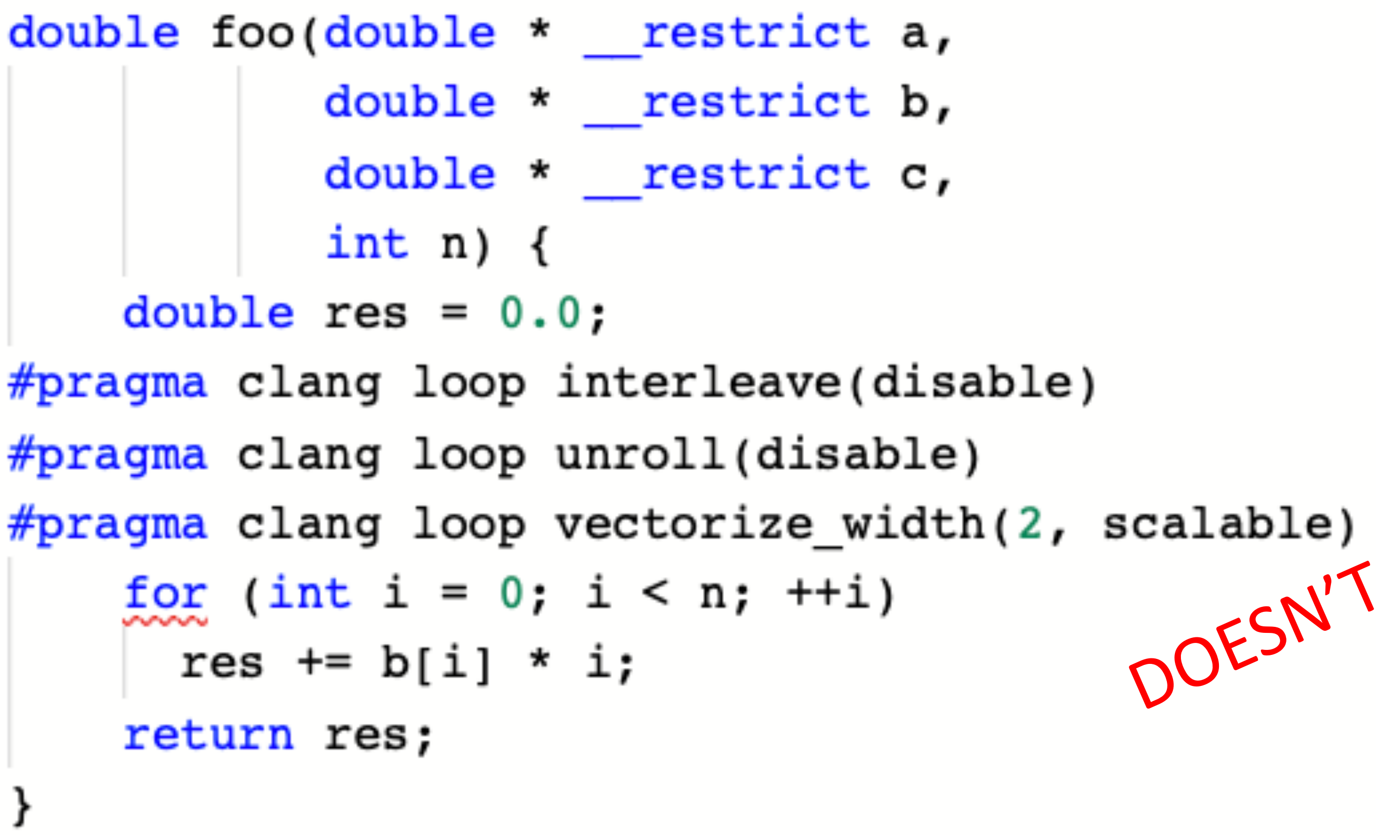
After the loop completes:



res = t[3] + t[2] + t[1] + t[0]

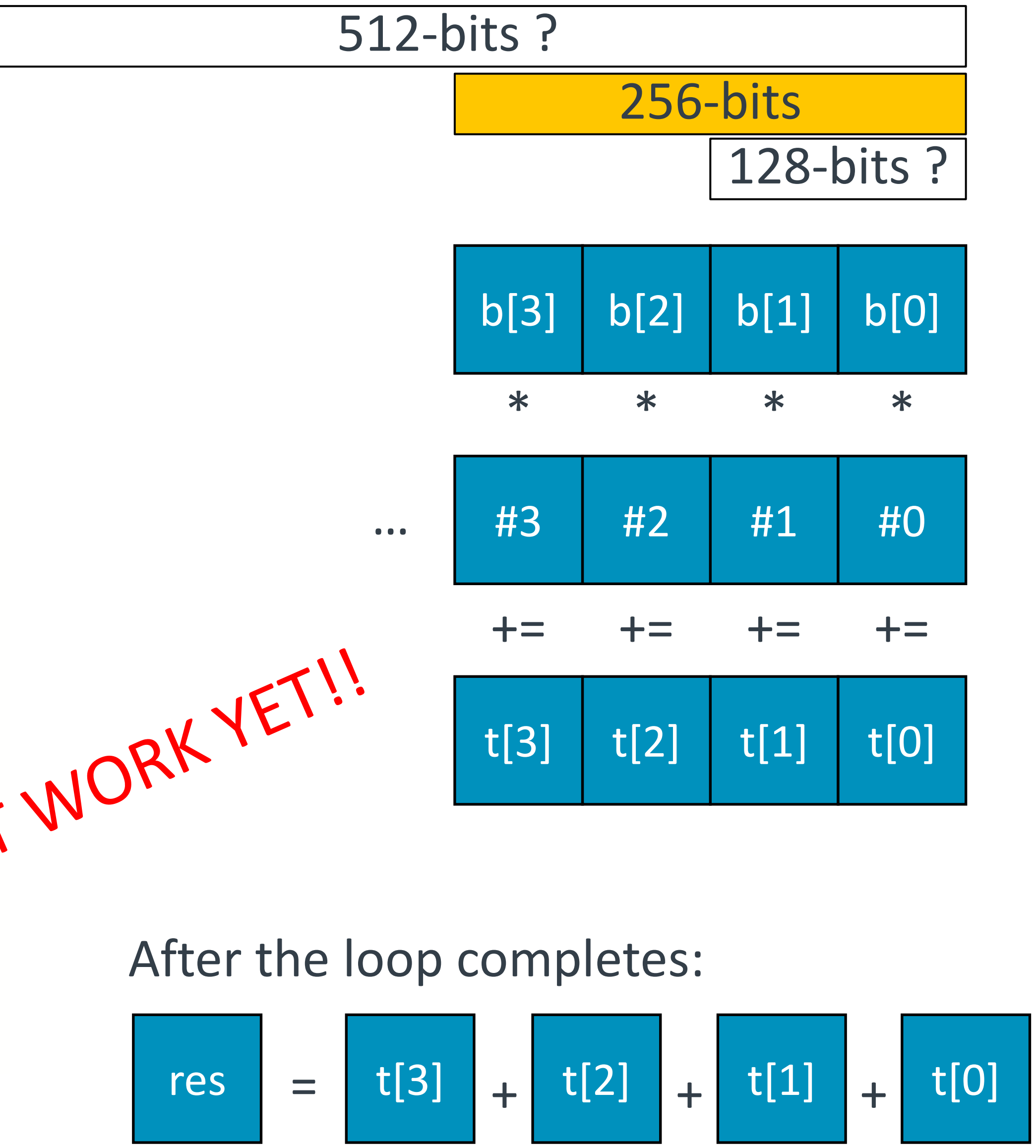
Example 9: Use induction variable

https://godbolt.org/z/vxaMcx



```
double foo(double * __restrict a,
           double * __restrict b,
           double * __restrict c,
           int n) {
    double res = 0.0;
#pragma clang loop interleave(disable)
#pragma clang loop unroll(disable)
#pragma clang loop vectorize_width(2, scalable)
    for (int i = 0; i < n; ++i)
      res += b[i] * i;
    return res;
}
```

DOESN'T WORK YET!!



After the loop completes:

Roadmap for SVE auto-vectorization

What does a production compiler need?

	LLVM 12	LLVM 13	LLVM 14
Safety	Source changes with pragma	Default-off flag for LNT	Default on?
Capability	~~1 loop~~ 32 loops	50 loops	Fully capable?
Quality	Terrible!	Improving	Good? Great?

Thank You
Danke
Gracias
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה