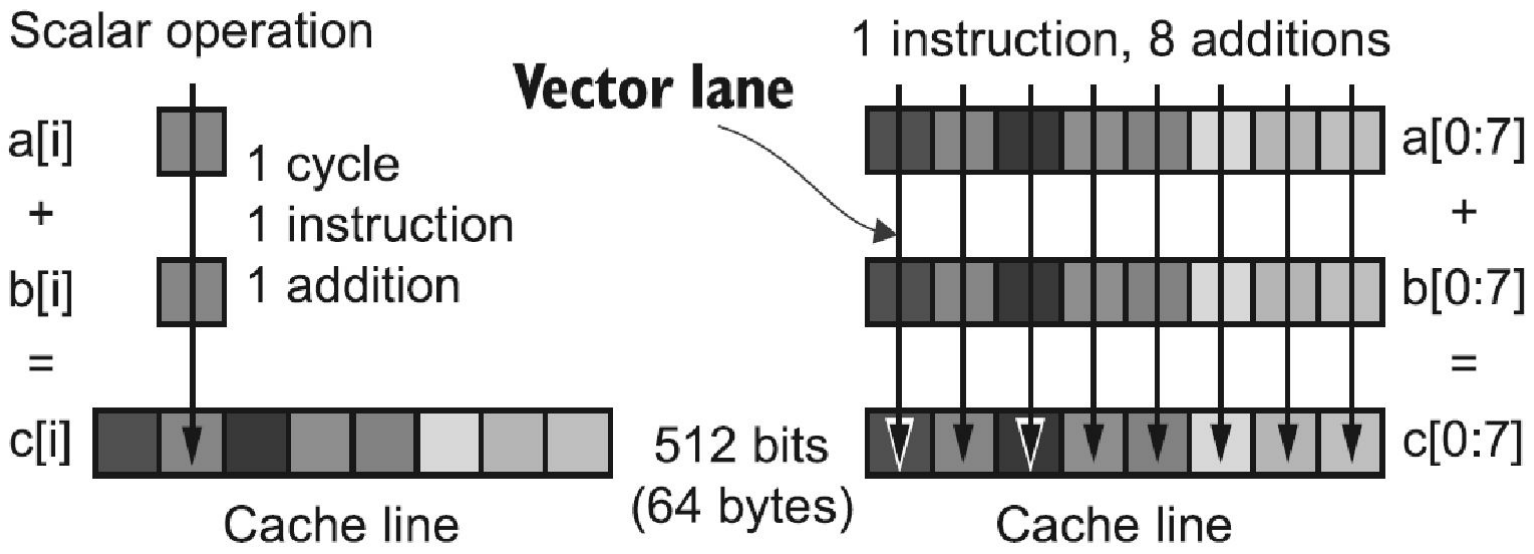
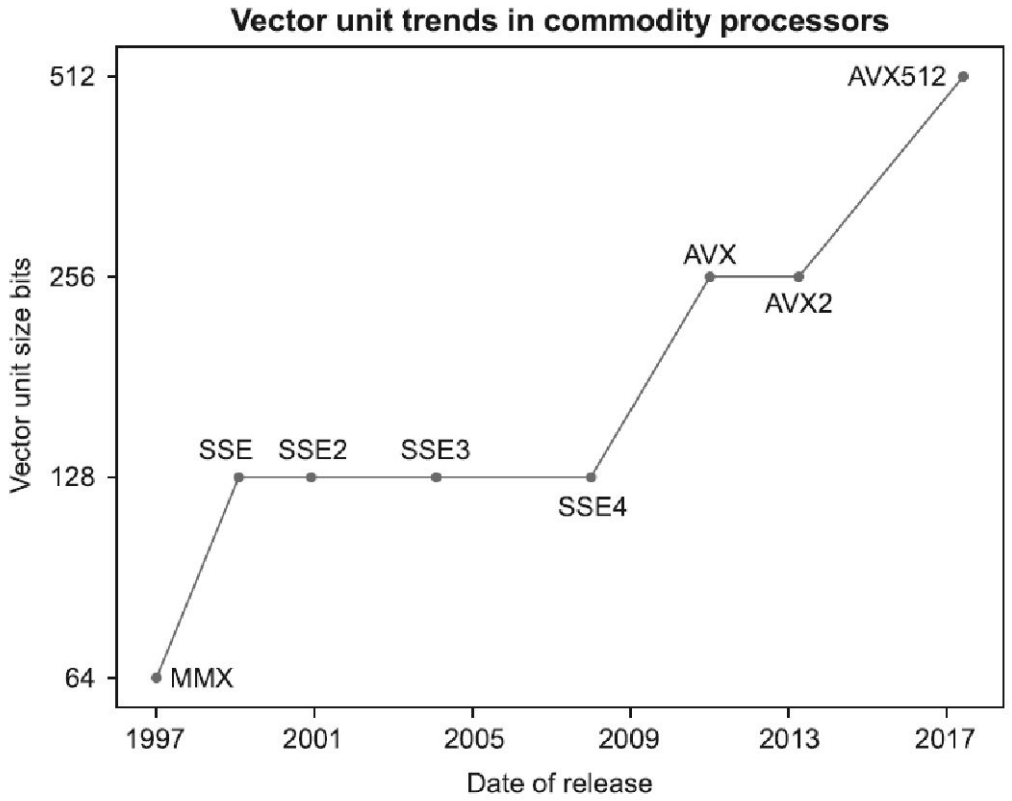
Vector instructions: SIMD

Single Instruction Multiple Data



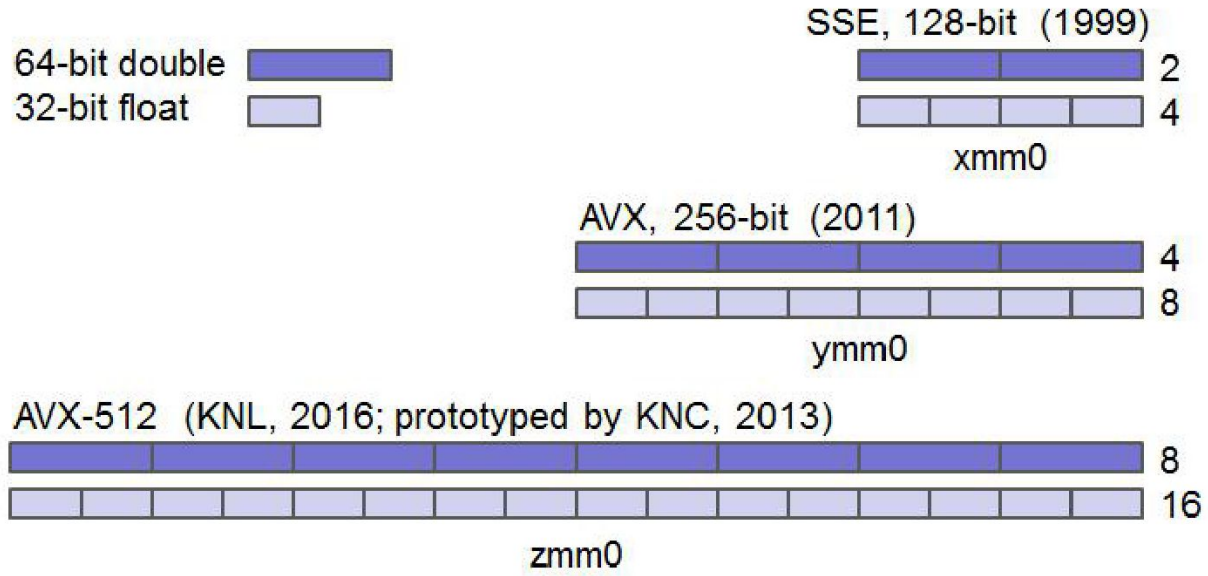
The SIMD instruction executes 8 DP floating point operations , corresponding to a whole cache line

Vectorization trend



SIMD instruction moved from 64 bits of MMX to 512 bits of AVX512

Vector size



A single SIMD register can be used with different data types

Vector register file

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

AVX512 registers (ZMM[0-31]) are asuperset of AVX/AVX2 YMM[0-15] registers, and of SSE XMM[0-15] registers

check for SIMD capabilities

```
└─[✓] lscpu | grep 'sse\|avx'
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdp
d aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer
_ppin cdp_l2 ssbd mba ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm rdt_a avx512f avx512dq
 xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local split_lock_detect user_shstk avx_vnni avx512_bf16 wbnoinvd dtherm ida arat pln pts hwp hwp_act_wind
```

Compiler flags

gcc/clang compilers can exploit vectorization instructions if the right flags are used

```
-msse
-msse2
...
-msse4.2
-mavx
-mavx2
-mavx512f
```

A full list of x86 instructions is available here: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

vector intrinsics types

the `<immintrin.h>` header provides intrinsinc instructions and types

INTEGER types

	types name	int 8bits	int 16bits	int 32bits	int 64bits
64bits	__m64	8x	4x	2x	1x
128bits	__m128i	16x	8x	4x	2x
256bits	__m256i	32x	16x	8x	4x
512bits	__m512i	64x	32x	16x	8x

FLOATING-POINT types

	types name	single precision	double precision
128bits	__m128	4x	-
	__m128d	-	2x
256bits	__m256	8x	-
	__m256d	-	4x
512bits	__m512	16x	-
	__m512d	-	8x

Intrinsic Name convention

The naming of SIMD instructions follows a pretty systematic set of rules:

```
_mm[bits]_[operation]_[type][modifier]
```

1. Prefix (`_mm`, `_mm256`, `_mm512`)

   `_mm` → 128-bit (xmm register)

   `_mm256` → 256-bit (ymm register)

   `_mm512` → 512-bit (zmm register)

2. Operation

   add, sub, mul, div, and, or, xor, etc.

   Sometimes longer: fma (fused multiply-add), cvtepi32_ps (convert int32 to float), etc.

Intrinsic Name convention

3. Type Suffix

- **ps** = packed single-precision float (float32)
- **pd** = packed double-precision float (float64)
- **epi8/16/32/64** = signed integers of given width
- **epu8/16/32/64** = unsigned integers of given width

4. Optional Modifiers

- **s =** scalar (lowest element only)
- **u =** unaligned load/store (e.g., _mm_loadu_si128)

Intrinsic Name examples

1. **`_mm256_add_ps`**
   `_mm256` → 256-bit vector (`ymm`)
   `add` → addition
   `ps` → packed single-precision floats

adds 8 floats in parallel.

2. **`_mm512_cvtepi32_pd`**
   `_mm512` → 512-bit (zmm)
   `cvt` → convert epi32 → from signed 32-bit integers
   `pd` → to double-precision floats

converts 8 int32 → 8 float64.

example

```
int main() {
    float a[N], b[N], c[N];
    // Initialize arrays
    for (int i = 0; i < N; i++) {
        a[i] = i * 1.0f;   b[i] = (N - i) * 1.0f;
    }
    // Perform vectorized addition using AVX
    for (int i = 0; i < N; i += 8) {
        // Load 8 floats from each array
        __m256 va = _mm256_load_ps(&a[i]);
        __m256 vb = _mm256_load_ps(&b[i]);
        // Add the two vectors
        __m256 vc = _mm256_add_ps(va, vb);
        // Store result into c
        _mm256_store_ps(&c[i], vc);
    }
```

GCC/Clang Extended Vector Types

There are many different ways of using vector operations in C++ code.

Many programmers use intrinsic instructions that provide very low-level access to the vector operations.

However, there is also a much more programmer-friendly approach: many modern C++ compilers support convenient **extended vector types**.

GCC/Clang Vector Types

This typedef let you write arithmetic on vector registers directly:

```
typedef float float8_t __attribute__ ((vector_size (8 * sizeof(float))));
```

The following lines are compiled to AVX instructions:

```
float8_t a, b, c;
c = a + b;   // compiler emits one SIMD add (e.g. AVX vaddps)
```

Warning: proper memory alignment needed

If we do dynamic memory allocation of `float8_t`, we **must make sure** that it points to a block of memory that is properly aligned.

Solutions:

- use `posix_memalign()` function;
- with C11 you can use `aligned_alloc()`
- with C17++ you can use

```
// compile with - std=c++17
std::vector<float8_t> x(10);
```

Code vectorization

Code vectorization

- The software changes required to exploit SIMD instructions are known as **code vectorization**.

- Initially, SIMD instructions were programmed in assembly. Later, special compiler intrinsics, which are small functions providing a one-to-one mapping to SIMD instructions, were introduced.

- ***Autovectorization****:* today all the major compilers support autovectorization for the popular processors, i.e., they can generate SIMD instructions straight from high-level code written in C/C++, Java, Rust, and other languages.

Code vectorization

However, in some cases, autovectorization does not succeed without intervention by a software engineer

- loop indices may *overflow*, preventing certain loop transformations.

- *pointers* in the program may overlap

- processors that don’t have efficient vector instructions for operations . (e.g. predicated operations) are not available on most processors.

Code vectorization

- Modern compilers can report whether a certain loop was vectorized. If the compiler cannot vectorize a loop, it is also able to tell the reason why it failed.

- Sometimes you can use compiler hints to drive vectorization

- When you cannot rely on compiler autovectorization or compiler hints, code can instead be written using *compiler intrinsics*. In most cases, compiler intrinsics provide a 1-to-1 mapping to assembly instructions.

Code vectorization

The vectorizer is usually structured in three phases:

- **Legality-check**: in this phase, the compiler checks if it is legal to transform the loop using vectors.
  - checks that the iterations of the loop are consecutive,
  - ensures that all memory and arithmetic operations can be widened into consecutive operations.
  - check that the order of operations will be preserved.

Code vectorization

Profitability-check: the vectorizer checks if a transformation is profitable.

- It compares different vectorization widths and figures out which one would be the fastest to execute.
- The vectorizer uses a cost model to predict the cost of different operations, such as scalar add or vector load.
- It takes into account the added instructions that shuffle data into registers, and estimate the cost of the loop guards that ensure that preconditions that allow vectorizations are met.

Code vectorization

Transformation: The vectorizer actually transforms the code

- This process also includes the insertion of guards that enable vectorization. If the loop use an unknown iteration count, the compiler has to generate a scalar version of the loop (remainder), in addition to the vectorized version of the loop, to handle the last few iterations.
- The compiler also has to check if pointers don't overlap, etc.

All of these transformations are done using information that is collected during the legality check phase.

Discovering Vectorization Opportunities

vectorization opportunity should start by analyzing hot loops in the program and checking why vectorization was not applied.

A good indicator for potential vectorization opportunities is a high **retiring rate** (above 80%).

Perhaps a workload executes a lot of simple instructions that can be replaced by vector instructions. In such situations, high retiring metric doesn’t translate into high performance.

Example 1: illegal vectorization

Listing 9.13 Vectorization: read-after-write dependence.

```
void vectorDependence(int *A, int n) {
  for (int i = 1; i < n; i++)
    A[i] = A[i-1] * 2;
}
```

This code cannot be vectorized due to read-after-write

```
$ clang  -Rpass-analysis=loop-vectorize  -march=native -O3 vectorDependence.c
```

```
Backward loop carried data dependence. Memory location is the same as accessed at vectorDependence.c:15:24 [-Rpass-analysis=loop-vectorize]
   15 |                 A[i] = A[i-1] * 2;
      |                        ^
```

Example 1: illegal vectorization

```
void vectorDependence_unrolled(int *A) {

    // First element is assumed initialized
    for (int i = 1; i+7 < N; i += 8) {
        A[i]   = A[i-1] * 2;
        A[i+1] = A[i-1] * 4;
        A[i+2] = A[i-1] * 8;
        A[i+3] = A[i-1] * 16;
        A[i+4] = A[i-1] * 32;
        A[i+5] = A[i-1] * 64;
        A[i+6] = A[i-1] * 128;
        A[i+7] = A[i-1] * 256;
    }
}
```

oes not work
penMP loop)

```
$ clang  -Rpass-analysis=loop-vectorize  -march=native -O3 vectorDependence.c
```

```
Backward loop carried data dependence. Memory location is the same as accessed at vectorDependence.c:27:18 [-Rpass-analysis=loop-vectorize]
   34 |         A[i+7] = A[i-1] * 256;
```

Example 1: illegal vectorization

```
void vectorDependence_unrolled_mm(int *A) {


    __m256i base = _mm256_set1_epi32(A[0]); // broadcast A[0] to 8 ints
    const int power2_const[8] = {2, 4, 8, 16, 32, 64, 128,256};
    __m256i power2 = _mm256_loadu_si256((__m256i*)power2_const);

    __m256i vals = _mm256_mullo_epi32(base, power2);
    _mm256_storeu_si256((__m256i*)&A[1], vals);
    base = _mm256_set1_epi32(256); // broadcast 256 to 8 ints
    for (int i = 9; i < N; i += 8) {
        // Compute the values by multiplying with A[i-8]
        vals = _mm256_mullo_epi32(vals, base);
        _mm256_storeu_si256((__m256i*)&A[i], vals);
    }
}
```

(dif tructions

```
$ clang  -Rpass-analysis=loop-vectorize  -march=native -O3 vectorDependence.c
```

```
vectorDependence.c:49:5: remark: loop not vectorized: loop control flow is not understood by vectorizer [-Rpass-analysis=loop-vectorize]
   49 |     for (int i = 9; i < N; i += 8) {
```

Example 1: illegal vectorization

```
void vectorDependence_unrolled_mm(int *A) {


    __m256i base = _mm256_set1_epi32(A[0]); // broadcast A[0] to 8 ints
    const int power2_const[8] = {2, 4, 8, 16, 32, 64, 128,256};
    __m256i power2 = _mm256_loadu_si256((__m256i*)power2_const);

    __m256i vals = _mm256_mullo_epi32(base, power2);
    _mm256_storeu_si256((__m256i*)&A[1], vals);
    base = _mm256_set1_epi32(256); // broadcast 256 to 8 ints
    for (int i = 9; i < N; i += 8) {
        // Compute the values by multiplying with A[i-8]
        vals = _mm256_mullo_epi32(vals, base);
        _mm256_storeu_si256((__m256i*)&A[i], vals);
    }
}
```

(dif tructions

```
$ clang  -Rpass-analysis=loop-vectorize  -march=native -O3 vectorDependence.c
```

```
vectorDependence.c:49:5: remark: loop not vectorized: loop control flow is not understood by vectorizer [-Rpass-analysis=loop-vectorize]
   49 |     for (int i = 9; i < N; i += 8) {
```

Why?

Example 1: illegal vectorization

```
void vectorDependence_vectorizable(int *A) {
    int base = A[0];
    for (int i = 1; i < N; i++) {
        A[i] = base << i;   // equivalent to A[0] * (2^i)
    }
}
```

rithm
IP loop)

```
$ clang  -Rpass-analysis=loop-vectorize  -march=native -O3 vectorDependence.c
```

```
vectorDependence.c:59:5: remark: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
```

Example 1: illegal vectorization

```
void vectorDependence_vectorizable(int *A) {
    int base = A[0];
    for (int i = 1; i < N; i++) {
        A[i] = base << i;   // equivalent to A[0] * (2^i)
    }
}
```

hm
loop)

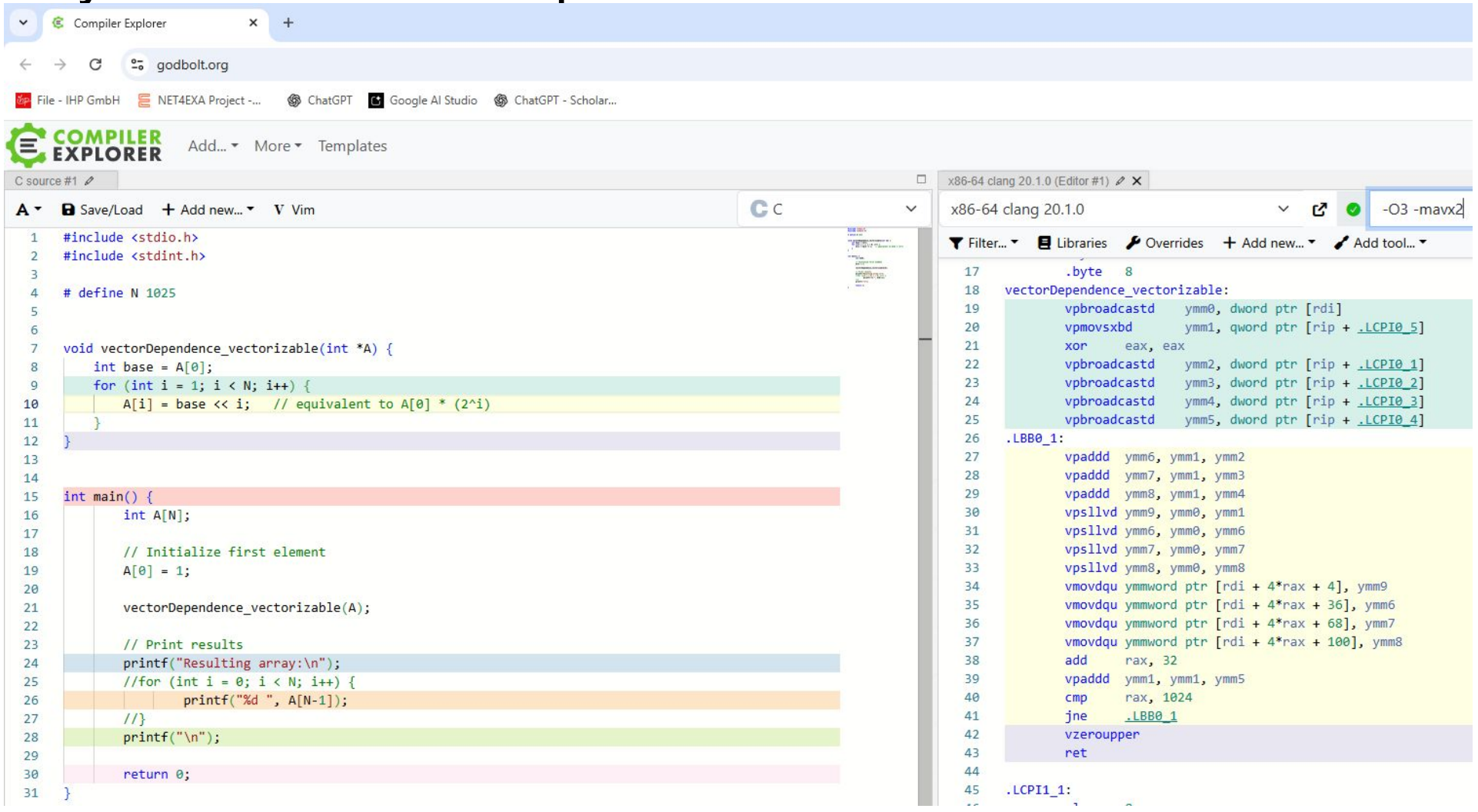
```
$ clang  -Rpass-analysis=loop-vectorize  -march=native -O3 vectorDependence.c
```

```
vectorDependence.c:59:5: remark: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
```

width:8 □ 8 integer in the same SIMD
interleaving:4 □ 4 SIMD instructions for each loop iteration

Example 1: illegal vectorization

Assembly of vectorized loop



Example 2: floating-point arithmetic

Listing 9.14 Vectorization: floating-point arithmetic.

```
1 // a.cpp
2 float calcSum(float* a, unsigned N) {
3   float sum = 0.0f;
4   for (unsigned i = 0; i < N; i++) {
5     sum += a[i];
6   }
7   return sum;
8 }
```

This code cannot be vectorized due to not associative FP o

```
$ clang  -Rpass-analysis=loop-vectorize  -march=native -O3 calcsum.c
```

```
calcsum.c:12:13: remark: loop not vectorized: cannot prove it is safe to reorder floating-point operations; allow reordering by specifying
pragma clang loop vectorize(enable)' before the loop or by providing the compiler option '-ffast-math' [-Rpass-analysis=loop-vectorize]
```

Example 2: floating-point arithmetic

Listing 9.14 Vectorization: floating-point arithmetic.

```
1 // a.cpp
2 float calcSum(float* a, unsigned N) {
3   float sum = 0.0f;
4   for (unsigned i = 0; i < N; i++) {
5     sum += a[i];
6   }
7   return sum;
8 }
```

This code cannot be vectorized due to not associative FP o

```
$ clang -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize  \
        -ffast-math -march=native -O3 calcsum.c
```

```
calcsum.c:11:5: remark: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
   11 |     for (unsigned i = 0; i < N; i++) {
      |     ^
calcsum.c:21:5: remark: vectorized loop (vectorization width: 8, interleaved count: 2) [-Rpass=loop-vectorize]
   21 |     for (unsigned i = 0; i < N; i++) {
      |     ^
calcsum.c:11:5: remark: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
   11 |     for (unsigned i = 0; i < N; i++) {
      |     ^
```

Example 3: memory overlap

Listing 9.15 a.c

```
1 void foo(float* a, float* b, float* c, unsigned N) {
2   for (unsigned i = 1; i < N; i++) {
3     c[i] = b[i];
4     a[i] = c[i-1];
5   }
6 }
```

When compilers cannot prove that a loop operates on arrays with non-overlapping memory regions, they usually choose to be on the safe side

```
$ clang -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize \
                    -march=native -O3 overlap.c
```

Example 3: memory overlap

Listing 9.15 a.c

```
1 void foo(float* a, float* b, float* c, unsigned N) {
2   for (unsigned i = 1; i < N; i++) {
3     c[i] = b[i];
4     a[i] = c[i-1];
5   }
6 }
```

When compilers cannot prove that a loop operates on arrays with non-overlapping memory regions, they usually choose to be on the safe side

```
$ clang -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize \
                    -march=native -O3 overlap.c
```

```
overlap.c:8:16: 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
Forward loop carried data dependence that prevents store-to-load forwarding. Memory location is the same as accessed at overlap.c:7:9 [-Rpa
ss-analysis=loop-vectorize]
    8 |         a[i] = c[i-1];
```

Example 3: memory overlap

Listing 5.7 a.c

```
1 void foo(float* __restrict__ a,
2          float* __restrict__ b,
3          float* __restrict__ c,
4          unsigned N) {
5   for (unsigned i = 1; i < N; i++) {
6     c[i] = b[i];
7     a[i] = c[i-1];
8   }
9 }
```

When a developer knows arrays a, b, and c do not overlap, it can insert the `__restrict__` keyword

```
$ clang -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize \
                    -march=native -O3 overlap.c
```

Example 3: memory overlap

Listing 5.7 a.c

```
1 void foo(float* __restrict__ a,
2          float* __restrict__ b,
3          float* __restrict__ c,
4          unsigned N) {
5   for (unsigned i = 1; i < N; i++) {
6     c[i] = b[i];
7     a[i] = c[i-1];
8   }
9 }
```

When a developer knows arrays a, b, and c do not overlap, it can insert the `__restrict__` keyword

```
$ clang -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize \
                    -march=native -O3 overlap.c
```

```
overlap.c:6:5: remark: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
    6 |     for (unsigned i = 1; i < N; i++) {
      |     ^
```

Example 4: Vectorization Is Not Beneficial

Listing 9.16 Vectorization: not beneficial.

```
1 // a.cpp
2 void stridedLoads(int *A, int *B, int n) {
3   for (int i = 0; i < n; i++)
4     A[i] += B[i * 3];
5 }
```

In some cases, the compiler can vectorize the loop but decide that doing so is not profitable

```
$ clang -c -O3 -mavx strideload.c -Rpass-missed=loop-vectorize
```

```
strideload.c:7:5: remark: the cost-model indicates that vectorization is not beneficial [-Rpass-missed=loop-vectorize]
    7 |     for (int i = 0; i < n; i++)
      |     ^
```

Programming style for better vectorization

- Use `__restrict__` attribute on pointers
- Use pragmas carefully:
  - to inform compiler
  - `#pragma unroll` can prevent loop vectorization
- avoid exceptions and break
- Use contiguous and aligned memory access (e.g. use SoA)
- Define local variables directly in the loop
- Avoid function call (use inlining)

Compiler flags

`-ftree-vectorize:` enables automatic vectorization of loops

`-march=native:` generate code optimized for the CPU of the machine you are compiling on

`-fopt-info-vec`: generate vectorization info.

`-fopt-info-vec-missed` : reports loops that couldn’t be vectorized.