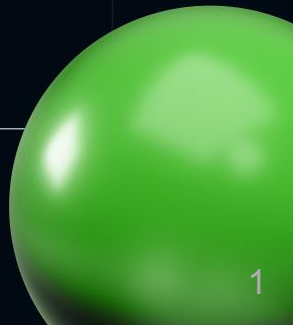


Loop unrolling in details



**Иван
Афанасьев**
Unipro



C++ Russia
2023



**Иван
Афанасьев**

Bio

- ММФ НГУ. к.ф.м.н.
- 12 лет C++
- 2 года в разработке
компиляторов 🧠

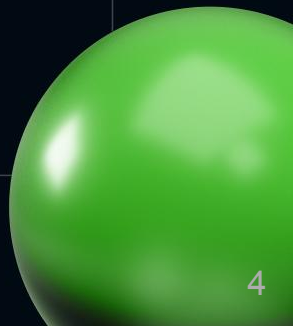
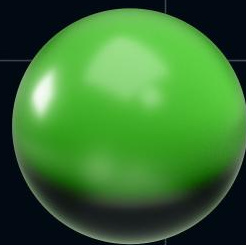
Outline of the talk

- Loop from compiler perspective
- Loop unrolling basics
- Loop unrolling overhead
- GCC and CLANG unroll details
- New optimization opportunities after unrolling
- Example



Outline of the talk

- Loop from compiler perspective
- Loop unrolling basics
- Loop unrolling overhead
- GCC and CLANG unroll details
- New optimization opportunities after unrolling
- Example



Loop from compiler perspective

```
for (i = s; i < b; i++)  
    statements(i);
```

Loop from compiler perspective

```
for (i = s; i < b; i++)  
    statements(i);
```

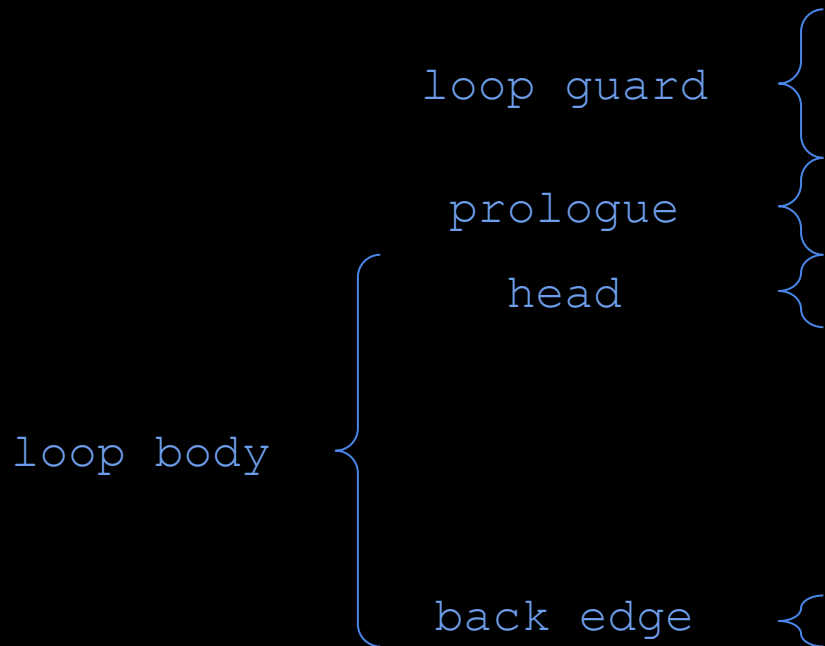
```
if (s < b) {  
    i = s;  
    do {  
        statements(i);  
        i++;  
    } while (i < b);  
}
```

Loop from compiler perspective

```
for (i = s; i < b; i++)  
    statements(i);
```

```
    bool cond = s < b;  
    if (!cond) goto LBB_EXIT;  
    i = s;  
LBB_HEAD:  
    statements(i);  
    ++i;  
    bool cond = i < b;  
    if (cond) goto LBB_HEAD;  
LBB_EXIT:
```

Loop from compiler perspective



```
bool cond = s < b;  
if (!cond) goto LBB_EXIT;  
i = s;  
LBB_HEAD:  
    statements(i);  
    ++i;  
    bool cond = i < b;  
    if (cond) goto LBB_HEAD;  
LBB_EXIT:
```


Loop payload and administration overhead

```
for (i = s; i < b; i++)  
    statements(i);
```

```
    bool cond = s < b;  
    if (!cond) goto LBB_EXIT;  
    i = s;  
LBB_HEAD:  
    statements(i);  
    ++i;  
    bool cond = i < b;  
    if (cond) goto LBB_HEAD;  
LBB_EXIT:
```

Loop payload and administration overhead

```
for (i = s; i < b; i++)  
    statements(i);
```

payload

```
bool cond = s < b;  
if (!cond) goto LBB_EXIT;  
i = s;  
LBB_HEAD:  
    statements(i);  
    ++i;  
    bool cond = i < b;  
    if (cond) goto LBB_HEAD;  
LBB_EXIT:
```

Loop payload and administration overhead

```
for (i = s; i < b; i++)  
    statements(i);
```

payload

loop administration
overhead

```
    bool cond = s < b;  
    if (!cond) goto LBB_EXIT;  
    i = s;  
  
LBB_HEAD:  
    statements(i);  
    ++i;  
    bool cond = i < b;  
    if (cond) goto LBB_HEAD;  
  
LBB_EXIT:
```

Loop payload and administration overhead

```
for (i = s; i < b; i++)  
    statements(i);
```

payload

loop administration
overhead



```
    bool cond = s < b;  
    if (!cond) goto LBB_EXIT;  
    i = s;  
  
LBB_HEAD:  
    statements(i);  
    ++i;  
    bool cond = i < b;  
    if (cond) goto LBB_HEAD;  
  
LBB_EXIT:
```

Examples. Loop

```
1 int sum(int *a, int n) {  
2     int x = 0;  
3     for (int i = 0; i < n; ++i)  
4         x += a[i];  
5     return x;  
6 }
```

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ 🖋 Add tool... ▾

```
1 sum(int*, int):  
2     test    esi, esi  
3     jle     .L4  
4     movsx   rsi, esi  
5     lea     rdx, [rdi+rsi*4]  
6     xor     eax, eax  
7 .L3:  
8     add     eax, DWORD PTR [rdi]  
9     add     rdi, 4  
10    cmp     rdi, rdx  
11    jne     .L3  
12    ret  
13 .L4:  
14    xor     eax, eax  
15    ret
```

Examples. Loop

TEST - Logical Compare

Computes the bitwise logical AND of the first operand and the second operand and sets the SF, ZF, and PF status flags.

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ 🛠 Add tool... ▾

```
1  sum(int*, int):  
2      test    esi, esi  
3      jle     .L4  
4      movsx   rsi, esi  
5      lea     rdx, [rdi+rsi*4]  
6      xor     eax, eax  
7  .L3:  
8      add     eax, DWORD PTR [rdi]  
9      add     rdi, 4  
10     cmp     rdi, rdx  
11     jne     .L3  
12     ret  
13  .L4:  
14     xor     eax, eax  
15     ret
```

Examples. Loop

JLE - Conditional Jump

Jump to the destination if one or more of the status flags is set (... ZF ...).

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ ✎ Add tool... ▾

```
1  sum(int*, int):  
2      test    esi, esi  
3      jle     .L4  
4      movsx   rsi, esi  
5      lea     rdx, [rdi+rsi*4]  
6      xor     eax, eax  
7  .L3:  
8      add     eax, DWORD PTR [rdi]  
9      add     rdi, 4  
10     cmp     rdi, rdx  
11     jne     .L3  
12     ret  
13  .L4:  
14     xor     eax, eax  
15     ret
```

Examples. Loop

LEA - Load Effective Address

Computes the effective address of the second operand and stores it in the first operand.

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ ✎ Add tool... ▾

```
1  sum(int*, int):  
2      test    esi, esi  
3      jle     .L4  
4      movsx   rsi, esi  
5      lea     rdx, [rdi+rsi*4]  
6      xor     eax, eax  
7  .L3:  
8      add     eax, DWORD PTR [rdi]  
9      add     rdi, 4  
10     cmp     rdi, rdx  
11     jne     .L3  
12     ret  
13  .L4:  
14     xor     eax, eax  
15     ret
```


Examples. Loop

XOR

Performs a bitwise XOR operation on the first and second operands and stores the result in the first operand.

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ 🛠 Add tool... ▾

```
1  sum(int*, int):  
2      test    esi, esi  
3      jle     .L4  
4      movsx   rsi, esi  
5      lea     rdx, [rdi+rsi*4]  
6      xor     eax, eax  
7  .L3:  
8      add     eax, DWORD PTR [rdi]  
9      add     rdi, 4  
10     cmp     rdi, rdx  
11     jne     .L3  
12     ret  
13  .L4:  
14     xor     eax, eax  
15     ret
```

Examples. Loop

```
1 int sum(int *a, int n) {  
2     int x = 0;  
3     for (int i = 0; i < n; ++i)  
4         x += a[i];  
5     return x;  
6 }
```

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ 🛠 Add tool... ▾

```
1 sum(int*, int):  
2     test    esi, esi  
3     jle     .L4  
4     movsx   rsi, esi  
5     lea     rdx, [rdi+rsi*4]  
6     xor     eax, eax  
7 .L3:  
8     add     eax, DWORD PTR [rdi]  
9     add     rdi, 4  
10    cmp     rdi, rdx  
11    jne     .L3  
12    ret  
13 .L4:  
14    xor     eax, eax  
15    ret
```

Examples. Loop

```
1 int sum(int *a, int n) {  
2     int x = 0;  
3     for (int i = 0; i < n; ++i)  
4         x += a[i];  
5     return x;  
6 }
```

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ 🖋 Add tool... ▾

```
1 sum(int*, int):  
2     test    esi, esi  
3     jle     .L4  
4     movsx   rsi, esi  
5     lea     rdx, [rdi+rsi*4]  
6     xor     eax, eax  
7 .L3:  
8     add     eax, DWORD PTR [rdi]  
9     add     rdi, 4  
10    cmp     rdi, rdx  
11    jne     .L3  
12    ret  
13 .L4:  
14    xor     eax, eax  
15    ret
```

loop guard

Examples. Loop

```
1 int sum(int *a, int n) {  
2     int x = 0;  
3     for (int i = 0; i < n; ++i)  
4         x += a[i];  
5     return x;  
6 }
```

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ 🖋 Add tool... ▾

```
1 sum(int*, int):  
2     test    esi, esi  
3     jle     .L4  
4     movsx   rsi, esi  
5     lea     rdx, [rdi+rsi*4]  
6     xor     eax, eax  
7 .L3:  
8     add     eax, DWORD PTR [rdi]  
9     add     rdi, 4  
10    cmp     rdi, rdx  
11    jne     .L3  
12    ret  
13 .L4:  
14    xor     eax, eax  
15    ret
```

loop guard

prologue

Examples. Loop

```
1 int sum(int *a, int n) {  
2     int x = 0;  
3     for (int i = 0; i < n; ++i)  
4         x += a[i];  
5     return x;  
6 }
```

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ 🖋 Add tool... ▾

```
1 sum(int*, int):  
2     test    esi, esi  
3     jle     .L4  
4     movsx   rsi, esi  
5     lea     rdx, [rdi+rsi*4]  
6     xor     eax, eax  
7 .L3:  
8     add     eax, DWORD PTR [rdi]  
9     add     rdi, 4  
10    cmp     rdi, rdx  
11    jne     .L3  
12    ret  
13 .L4:  
14    xor     eax, eax  
15    ret
```

loop guard

prologue

iteration check back edge

Examples. Loop

```
1 int sum(int *a, int n) {  
2     int x = 0;  
3     for (int i = 0; i < n; ++i)  
4         x += a[i];  
5     return x;  
6 }
```

A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries + Add new... ▾ 🖋 Add tool... ▾

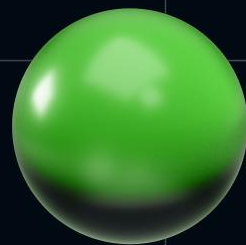
```
1 sum(int*, int):  
2     test    esi, esi  
3     jle     .L4  
4     movsx   rsi, esi  
5     lea     rdx, [rdi+rsi*4]  
6     xor     eax, eax  
7 .L3:  
8     add     eax, DWORD PTR [rdi]  
9     add     rdi, 4  
10    cmp     rdi, rdx  
11    jne     .L3  
12    ret  
13 .L4:  
14    xor     eax, eax  
15    ret
```

Annotations:

- Lines 2-3: **loop guard**
- Lines 4-6: **prologue**
- Line 8: **payload**
- Lines 9-11: **iteration check back edge**

Outline of the talk

- Loop from compiler perspective
- **Loop unrolling basics**
- Loop unrolling overhead
- GCC and CLANG unroll details
- New optimization opportunities after unrolling
- Example



Loop unrolling

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```


Loop unrolling

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

```
N_jumps /= 4  
N_checks /= 4
```

Loop unrolling

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

`N_jumps /= 4`

`N_checks /= 4`

`(b - s) % 4 == 0 ?`

Prolog + epilogue loop

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

Overhead:

- Epilogue (remainder) loop
- Calculate b'

Prolog + epilogue loop

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

Overhead:

- Epilogue (remainder) loop
- Calculate b'

Prolog + epilogue loop

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

Overhead:

- Epilogue (remainder) loop
- Calculate b'

Prolog + epilogue loop

```
b' = b - (b - s) % 4;
for (i = s; i < b'; i += 4) {
    statements(i);
    statements(i + 1);
    statements(i + 2);
    statements(i + 3);
}
for (i = b'; i < b; i++)
    statements(i);
```

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$v1 = b - s$

$v2 = v1 \% 4$

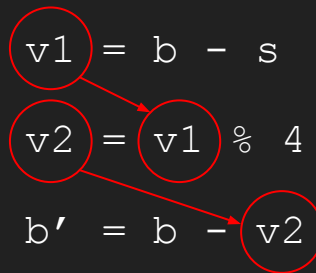
$b' = b - v2$

Prolog + epilogue loop

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

Overhead:

- Epilogue (remainder) loop
- Calculate b'



stall!

(latency)

Prolog + epilogue loop

```
b' = b - (b - s) % 4;
for (i = s; i < b'; i += 4) {
    statements(i);
    statements(i + 1);
    statements(i + 2);
    statements(i + 3);
}
for (i = b'; i < b; i++)
    statements(i);
```

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$v1 = b - s$

$v2 = v1 \% 4$

$b' = b - v2$

Prolog + epilogue loop

- $V1 \% UNROLLED_STEP$ is expensive in general case
- But ...

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$v1 = b - s$

$v2 = v1 \% 4$

$b' = b - v2$

Prolog + epilogue loop

- `V1 % UNROLLED_STEP` is expensive in general case
- But ...
 - If `V1 >= 0`

Overhead:

- Epilogue (remainder) loop
- Calculate `b'`

`v1 = b - s`

`v2 = v1 % 4`

`b' = b - v2`

Prolog + epilogue loop

- $V1 \% UNROLLED_STEP$ is expensive in general case
- But ...
 - If $V1 \geq 0$
 - If $UNROLLED_STEP == 2**K$

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$$v1 = b - s$$

$$v2 = v1 \% 4$$

$$b' = b - v2$$

Prolog + epilogue loop

- $V1 \% UNROLLED_STEP$ is expensive in general case
- But ...
 - If $V1 \geq 0$
 - If $UNROLLED_STEP == 2 * K$

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$$v1 = b - s$$

$$v2 = v1 \% 4$$

$$b' = b - v2$$

V1	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?
2*K	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Prolog + epilogue loop

- $V1 \% UNROLLED_STEP$ is expensive in general case
- But ...
 - If $V1 \geq 0$
 - If $UNROLLED_STEP == 2 * K$

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$$v1 = b - s$$

$$v2 = v1 \% 4$$

$$b' = b - v2$$

V1	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?
2*K	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Prolog + epilogue loop

- $V1 \% \text{UNROLLED_STEP}$ is expensive in general case
- But ...
 - If $V1 \geq 0$
 - If $\text{UNROLLED_STEP} == 2 \times K$

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$$v1 = b - s$$

$$v2 = v1 \% 4$$

$$b' = b - v2$$

V1	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?
$2 \times K$	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
$2 \times K - 1$	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

Prolog + epilogue loop

- $V1 \% UNROLLED_STEP$ is expensive in general case
- But ...
 - If $V1 \geq 0$
 - If $UNROLLED_STEP == 2 * K$

$V1 \% 2 * K == V1 \& (2 * K - 1)$

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$v1 = b - s$

$v2 = v1 \% 4$

$b' = b - v2$

V1	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?
2*K	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
2*K - 1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

Prolog + epilogue loop

- If `s == 0`
 for (`i = 0; i < b; ++i`)

Overhead:

- Epilogue (remainder) loop
- Calculate `b'`

$$b' = b - (b - s) \% 4$$

Prolog + epilogue loop

- If `s == 0`
 `for (i = 0; i < b; ++i)`
- If `b >= 0`

Overhead:

- Epilogue (remainder) loop
- Calculate `b'`

$$b' = b - (b - s) \% 4$$

b

0

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

?

Prolog + epilogue loop

- If `s == 0`
 `for (i = 0; i < b; ++i)`
- If `b >= 0`
- If `UNROLLED_STEP == 2**K`

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$$b' = b - (b - s) \% 4$$

b	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?
2**K	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Prolog + epilogue loop

- If $s == 0$
 for ($i = 0$; $i < b$; $++i$)
- If $b \geq 0$
- If $UNROLLED_STEP == 2**K$
- $b' = b - b \% 2**K$

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$$b' = b - (b - s) \% 4$$

b	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?
2**K	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
b'	0	?	?	?	?	?	?	?	?	?	0	0	0	0	0

Prolog + epilogue loop

- If $s == 0$
 for ($i = 0$; $i < b$; $++i$)
- If $b \geq 0$
- If $UNROLLED_STEP == 2**K$
- $b' = b - b \% 2**K$

$b - b \% 2**K == b \ \& \ \sim(2**K - 1)$

Overhead:

- Epilogue (remainder) loop
- Calculate b'

$$b' = b - (b - s) \% 4$$

b	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?
2**K	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
b'	0	?	?	?	?	?	?	?	?	?	0	0	0	0	0
$\sim(2**K - 1)$	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0

Prolog + epilogue loop

- If $s == 0$
 for ($i = 0$; $i < b$; $++i$)
- If $b \geq 0$
- If $\text{UNROLLED_STEP} == 2**K$
- $b' = b - b \% 2**K$

$b - b \% 2**K == b \ \& \ \sim(2**K - 1)$

Overhead:

- Epilogue (remainder) loop
- Calculate b'

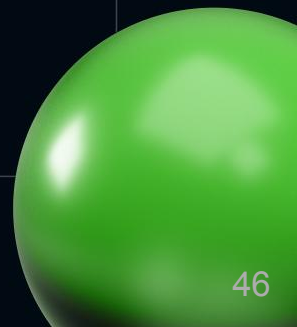
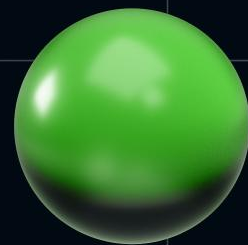
$$b' = b - (b - s) \% 4$$

b	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?
2**K	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
b'	0	?	?	?	?	?	?	?	?	?	0	0	0	0	0
$\sim(2**K - 1)$	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0

single
instruction
prologue!

Outline of the talk

- Loop from compiler perspective
- Loop unrolling basics
- **Loop unrolling overhead**
- GCC and CLANG unroll details
- New optimization opportunities after unrolling
- Example



origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```


origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 100$
Overhead:

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 100$
Overhead:

```
N_jumps   = 99  
N_checks = 101
```

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 100$
Overhead:

```
N_jumps   = 99  
N_checks  = 101
```

```
N_jumps   = 25  
N_checks  = 27  
extra prologue
```

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 100$
Overhead:

```
N_jumps   = 99  
N_checks = 101
```

```
N_jumps   = 25  
N_checks = 27  
extra prologue
```

```
N_jumps   = 24  
N_checks = 26
```

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 100$
Overhead:

```
N_jumps  = 99  
N_checks = 101
```

```
N_jumps  = 25  
N_checks = 27  
extra prologue
```



```
N_jumps  = 24  
N_checks = 26
```



origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 4$

Overhead:

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 4$
Overhead:

```
N_jumps   = 3  
N_checks  = 5
```


origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 4$
Overhead:

```
N_jumps   = 3  
N_checks  = 5
```

```
N_jumps   = 1  
N_checks  = 4  
extra prologue
```

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 4$
Overhead:

```
N_jumps   = 3  
N_checks  = 5
```

```
N_jumps   = 1  
N_checks  = 4  
extra prologue
```

```
N_jumps   = 0  
N_checks  = 2
```

origin loop

```
for (i = s; i < b; i++)  
    statements(i);
```

unroll + runtime remainder

```
b' = b - (b - s) % 4;  
for (i = s; i < b'; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}  
for (i = b'; i < b; i++)  
    statements(i);
```

unroll + no remainder guarantee

```
for (i = s; i < b; i += 4) {  
    statements(i);  
    statements(i + 1);  
    statements(i + 2);  
    statements(i + 3);  
}
```

Case: $b - s == 4$
Overhead:

```
N_jumps   = 3  
N_checks  = 5
```



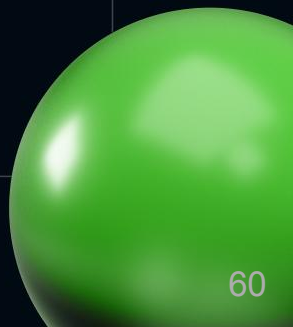
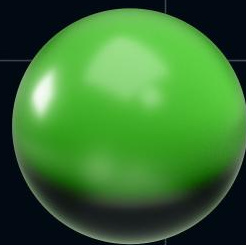
```
N_jumps   = 1  
N_checks  = 4  
extra prologue
```

```
N_jumps   = 0  
N_checks  = 2
```



Outline of the talk

- Loop from compiler perspective
- Loop unrolling basics
- Loop unrolling overhead
- **GCC and CLANG unroll details**
- New optimization opportunities after unrolling
- Example



Outline of the talk

- Loop from compiler perspective
- Loop unrolling basics
- Loop unrolling overhead
- **GCC and CLANG unroll details**
- New optimization opportunities after unrolling
- Example

GCC 12.2

-O2
-march=skylake
-fno-tree-vectorize

CLANG 15.0

-O2
-march=skylake
-fno-vectorize
-fno-slp-vectorize

Examples. Loop (GCC vs CLANG)

```
1 int sum(int *a, int n) {  
2     int x = 0;  
3     for (int i = 0; i < n; ++i)  
4         x += a[i];  
5     return x;  
6 }  
7
```

Examples. Loop (GCC vs CLANG)

```
x86-64 clang 15.0.0 -O2 -march=skylake -fno-vectorize -fno-  
A Output... Filter... Libraries + Add new... Add tool...  
1 sum(int*, int): # @sum(int*, int)  
2     test     esi, esi  
3     jle      .LBB0_1  
4     mov     esi, esi  
5     lea     rax, [rsi - 1]  
6     mov     ecx, esi  
7     and     ecx, 7  
8     cmp     rax, 7  
9     jae     .LBB0_8  
10    xor     edx, edx  
11    xor     eax, eax  
12    jmp     .LBB0_4  
13    .LBB0_1:  
14    xor     eax, eax  
15    ret  
16    .LBB0_8:  
17    and     esi, -8  
18    xor     edx, edx  
19    xor     eax, eax  
20    .LBB0_9: # =>This Inner Loop Header: Depth=1  
21    add     eax, dword ptr [rdi + 4*rdx]  
22    add     eax, dword ptr [rdi + 4*rdx + 4]  
23    add     eax, dword ptr [rdi + 4*rdx + 8]  
24    add     eax, dword ptr [rdi + 4*rdx + 12]  
25    add     eax, dword ptr [rdi + 4*rdx + 16]  
26    add     eax, dword ptr [rdi + 4*rdx + 20]  
27    add     eax, dword ptr [rdi + 4*rdx + 24]  
28    add     eax, dword ptr [rdi + 4*rdx + 28]  
29    add     rdx, 8  
30    cmp     rsi, rdx  
31    jne     .LBB0_9  
32    .LBB0_4:  
33    test    rcx, rcx  
34    je      .LBB0_7  
35    lea     rdx, [rdi + 4*rdx]  
36    xor     esi, esi  
37    .LBB0_6: # =>This Inner Loop Header: Depth=1  
38    add     eax, dword ptr [rdx + 4*rsi]  
39    inc     rsi  
40    cmp     rcx, rsi  
41    jne     .LBB0_6  
42    .LBB0_7:  
43    ret
```

Examples. Loop (GCC vs CLANG)

```
x86-64 clang 15.0.0 -O2 -march=skylake -fno-vectorize -fno-  
A Output... Filter... Libraries + Add new... Add tool...  
1 sum(int*, int): # @sum(int*, int)  
2     test    esi, esi  
3     jle     .LBB0_1  
4     mov     esi, esi  
5     lea     rax, [rsi - 1]  
6     mov     ecx, esi  
7     and     ecx, 7  
8     cmp     rax, 7  
9     jae     .LBB0_8  
10    xor     edx, edx  
11    xor     eax, eax  
12    jmp     .LBB0_4  
13 .LBB0_1:  
14    xor     eax, eax  
15    ret  
16 .LBB0_8:  
17    and     esi, -8  
18    xor     edx, edx  
19    xor     eax, eax  
20 .LBB0_9: # =>This Inner Loop Header: Depth=1  
21    add     eax, dword ptr [rdi + 4*rdx]  
22    add     eax, dword ptr [rdi + 4*rdx + 4]  
23    add     eax, dword ptr [rdi + 4*rdx + 8]  
24    add     eax, dword ptr [rdi + 4*rdx + 12]  
25    add     eax, dword ptr [rdi + 4*rdx + 16]  
26    add     eax, dword ptr [rdi + 4*rdx + 20]  
27    add     eax, dword ptr [rdi + 4*rdx + 24]  
28    add     eax, dword ptr [rdi + 4*rdx + 28]  
29    add     rdx, 8  
30    cmp     rsi, rdx  
31    jne     .LBB0_9  
32 .LBB0_4:  
33    test    rcx, rcx  
34    je      .LBB0_7  
35    lea     rdx, [rdi + 4*rdx]  
36    xor     esi, esi  
37 .LBB0_6: # =>This Inner Loop Header: Depth=1  
38    add     eax, dword ptr [rdx + 4*rsi]  
39    inc     rsi  
40    cmp     rcx, rsi  
41    jne     .LBB0_6  
42 .LBB0_7:  
43    ret
```

loop guard

loop guard

Examples. Loop (GCC vs CLANG)

```
x86-64 clang 15.0.0 -O2 -march=skylake -fno-vectorize -fno-  
A Output... Filter... Libraries + Add new... Add tool...  
1 sum(int*, int): # @sum(int*, int)  
2     test    esi, esi  
3     jle     .LBB0_1  
4     mov     esi, esi  
5     lea     rax, [rsi - 1]  
6     mov     ecx, esi  
7     and     ecx, 7  
8     cmp     rax, 7  
9     jae     .LBB0_8  
10    xor     edx, edx  
11    xor     eax, eax  
12    jmp     .LBB0_4  
13 .LBB0_1:  
14    xor     eax, eax  
15    ret  
16 .LBB0_8:  
17    and     esi, -8  
18    xor     edx, edx  
19    xor     eax, eax  
20 .LBB0_9: # =>This Inner Loop Header: Depth=1  
21    add     eax, dword ptr [rdi + 4*rdx]  
22    add     eax, dword ptr [rdi + 4*rdx + 4]  
23    add     eax, dword ptr [rdi + 4*rdx + 8]  
24    add     eax, dword ptr [rdi + 4*rdx + 12]  
25    add     eax, dword ptr [rdi + 4*rdx + 16]  
26    add     eax, dword ptr [rdi + 4*rdx + 20]  
27    add     eax, dword ptr [rdi + 4*rdx + 24]  
28    add     eax, dword ptr [rdi + 4*rdx + 28]  
29    add     rdx, 8  
30    cmp     rsi, rdx  
31    jne     .LBB0_9  
32 .LBB0_4:  
33    test    rcx, rcx  
34    je      .LBB0_7  
35    lea     rdx, [rdi + 4*rdx]  
36    xor     esi, esi  
37 .LBB0_6: # =>This Inner Loop Header: Depth=1  
38    add     eax, dword ptr [rdx + 4*rsi]  
39    inc     rsi  
40    cmp     rcx, rsi  
41    jne     .LBB0_6  
42 .LBB0_7:  
43    ret
```

loop guard

p1 x8

loop guard

p1

Examples. Loop (GCC vs CLANG)

```
x86-64 clang 15.0.0 -O2 -march=skylake -fno-vectorize -fno-  
A Output... Filter... Libraries + Add new... Add tool...  
1 sum(int*, int): # @sum(int*, int)  
2     test    esi, esi  
3     jle     .LBB0_1  
4     mov     esi, esi  
5     lea     rax, [rsi - 1]  
6     mov     ecx, esi  
7     and     ecx, 7  
8     cmp     rax, 7  
9     jae     .LBB0_8  
10    xor     edx, edx  
11    xor     eax, eax  
12    jmp     .LBB0_4  
13 .LBB0_1:  
14    xor     eax, eax  
15    ret  
16 .LBB0_8:  
17    and     esi, -8  
18    xor     edx, edx  
19    xor     eax, eax  
20 .LBB0_9: # =>This Inner Loop Header: Depth=1  
21    add     eax, dword ptr [rdi + 4*rdx]  
22    add     eax, dword ptr [rdi + 4*rdx + 4]  
23    add     eax, dword ptr [rdi + 4*rdx + 8]  
24    add     eax, dword ptr [rdi + 4*rdx + 12]  
25    add     eax, dword ptr [rdi + 4*rdx + 16]  
26    add     eax, dword ptr [rdi + 4*rdx + 20]  
27    add     eax, dword ptr [rdi + 4*rdx + 24]  
28    add     eax, dword ptr [rdi + 4*rdx + 28]  
29    add     rdx, 8  
30    cmp     rsi, rdx  
31    jne     .LBB0_9  
32 .LBB0_4:  
33    test    rcx, rcx  
34    je      .LBB0_7  
35    lea     rdx, [rdi + 4*rdx]  
36    xor     esi, esi  
37 .LBB0_6: # =>This Inner Loop Header: Depth=1  
38    add     eax, dword ptr [rdx + 4*rsi]  
39    inc     rsi  
40    cmp     rcx, rsi  
41    jne     .LBB0_6  
42 .LBB0_7:  
43    ret
```

Annotations in the image:

- loop guard** (red text) pointing to lines 2-3: `test esi, esi` and `jle .LBB0_1`.
- pl x8** (green text) pointing to the inner loop body (lines 21-28).
- iter x8 check + be** (red text) pointing to lines 29-31: `add rdx, 8`, `cmp rsi, rdx`, and `jne .LBB0_9`.
- loop guard** (red text) pointing to lines 33-34: `test rcx, rcx` and `je .LBB0_7`.
- pl** (green text) pointing to the second inner loop body (lines 38-41).
- iter check + be** (red text) pointing to lines 39-41: `inc rsi`, `cmp rcx, rsi`, and `jne .LBB0_6`.

Examples. Loop (GCC vs CLANG)

x86-64 clang 15.0.0 -O2 -march=skylake -fno-vectorize -fno-

A Output... Filter... Libraries + Add new... Add tool...

```
1 sum(int*, int):                                # @sum(int*, int)
2     test     esi, esi
3     jle      .LBB0_1
4     mov     esi, esi
5     lea     rax, [rsi - 1]
6     mov     ecx, esi
7     and     ecx, 7
8     cmp     rax, 7
9     jae     .LBB0_8
10    xor     edx, edx
11    xor     eax, eax
12    jmp     .LBB0_4
13 .LBB0_1:
14    xor     eax, eax
15    ret
16 .LBB0_8:
17    and     esi, -8
18    xor     edx, edx
19    xor     eax, eax
20 .LBB0_9:                                # =>This Inner Loop Header: Depth=1
21    add     eax, dword ptr [rdi + 4*rdx]
22    add     eax, dword ptr [rdi + 4*rdx + 4]
23    add     eax, dword ptr [rdi + 4*rdx + 8]
24    add     eax, dword ptr [rdi + 4*rdx + 12]
25    add     eax, dword ptr [rdi + 4*rdx + 16]
26    add     eax, dword ptr [rdi + 4*rdx + 20]
27    add     eax, dword ptr [rdi + 4*rdx + 24]
28    add     eax, dword ptr [rdi + 4*rdx + 28]
29    add     rdx, 8
30    cmp     rsi, rdx
31    jne     .LBB0_9
32 .LBB0_4:
33    test     rcx, rcx
34    je      .LBB0_7
35    lea     rdx, [rdi + 4*rdx]
36    xor     esi, esi
37 .LBB0_6:                                # =>This Inner Loop Header: Depth=1
38    add     eax, dword ptr [rdx + 4*rsi]
39    inc     rsi
40    cmp     rcx, rsi
41    jne     .LBB0_6
42 .LBB0_7:
43    ret
```

loop guard

prologue

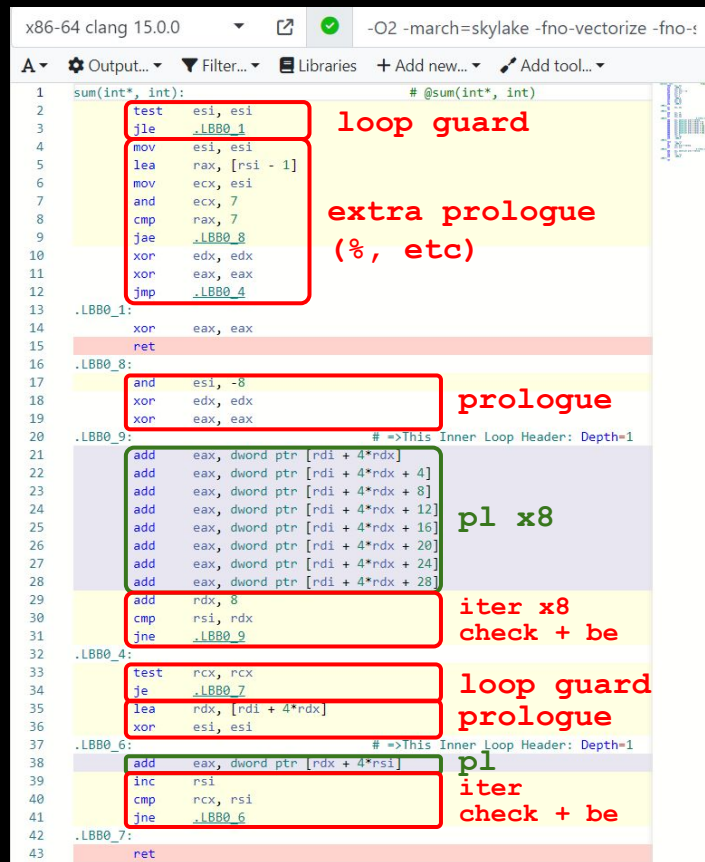
pl x8

iter x8
check + be

loop guard
prologue

pl
iter
check + be

Examples. Loop (GCC vs CLANG)



```
x86-64 clang 15.0.0 -O2 -march=skylake -fno-vectorize -fno-...
A Output... Filter... Libraries + Add new... Add tool...

1 sum(int*, int):                                #@sum(int*, int)
2     test     esi, esi
3     jle      .LBB0_1
4     mov      esi, esi
5     lea      rax, [rsi - 1]
6     mov      ecx, esi
7     and      ecx, 7
8     cmp      rax, 7
9     jae      .LBB0_8
10    xor      edx, edx
11    xor      eax, eax
12    jmp      .LBB0_4
13
.LBB0_1:
14    xor      eax, eax
15    ret
16
.LBB0_8:
17    and      esi, -8
18    xor      edx, edx
19    xor      eax, eax
20
.LBB0_9:                                          # =>This Inner Loop Header: Depth=1
21    add      eax, dword ptr [rdi + 4*rdx]
22    add      eax, dword ptr [rdi + 4*rdx + 4]
23    add      eax, dword ptr [rdi + 4*rdx + 8]
24    add      eax, dword ptr [rdi + 4*rdx + 12]
25    add      eax, dword ptr [rdi + 4*rdx + 16]
26    add      eax, dword ptr [rdi + 4*rdx + 20]
27    add      eax, dword ptr [rdi + 4*rdx + 24]
28    add      eax, dword ptr [rdi + 4*rdx + 28]
29    add      rdx, 8
30    cmp      rsi, rdx
31    jne      .LBB0_9
32
.LBB0_4:
33    test     rcx, rcx
34    je       .LBB0_7
35    lea      rdx, [rdi + 4*rdx]
36    xor      esi, esi
37
.LBB0_6:                                          # =>This Inner Loop Header: Depth=1
38    add      eax, dword ptr [rdx + 4*rsi]
39    inc      rsi
40    cmp      rcx, rsi
41    jne      .LBB0_6
42
.LBB0_7:
43    ret
```

Annotations in the image:

- loop guard**: Points to lines 2-3 (`test esi, esi`; `jle .LBB0_1`).
- extra prologue (% , etc)**: Points to lines 4-12, covering the initial setup and jump.
- prologue**: Points to lines 17-19 (`and esi, -8`; `xor edx, edx`; `xor eax, eax`).
- pl x8**: Points to the inner loop body (lines 21-28), indicating a prologue of 8 iterations.
- iter x8 check + be**: Points to lines 29-31 (`add rdx, 8`; `cmp rsi, rdx`; `jne .LBB0_9`).
- loop guard prologue**: Points to lines 33-36 (`test rcx, rcx`; `je .LBB0_7`; `lea rdx, [rdi + 4*rdx]`; `xor esi, esi`).

Examples. Loop (GCC vs CLANG)

What about
GCC ?

```
x86-64 clang 15.0.0 -O2 -march=skylake -fno-vectorize -fno-  
A Output... Filter... Libraries + Add new... Add tool...  
1 sum(int*, int): # @sum(int*, int)  
2     test     esi, esi  
3     jle      .LBB0_1  
4     mov     esi, esi  
5     lea     rax, [rsi - 1]  
6     mov     ecx, esi  
7     and     ecx, 7  
8     cmp     rax, 7  
9     jae     .LBB0_8  
10    xor     edx, edx  
11    xor     eax, eax  
12    jmp     .LBB0_4  
13  
14    .LBB0_1:  
15    xor     eax, eax  
16    ret  
17  
18    .LBB0_8:  
19    and     esi, -8  
20    xor     edx, edx  
21    xor     eax, eax  
22    .LBB0_9: # =>This Inner Loop Header: Depth=1  
23    add     eax, dword ptr [rdi + 4*rdx]  
24    add     eax, dword ptr [rdi + 4*rdx + 4]  
25    add     eax, dword ptr [rdi + 4*rdx + 8]  
26    add     eax, dword ptr [rdi + 4*rdx + 12]  
27    add     eax, dword ptr [rdi + 4*rdx + 16]  
28    add     eax, dword ptr [rdi + 4*rdx + 20]  
29    add     eax, dword ptr [rdi + 4*rdx + 24]  
30    add     eax, dword ptr [rdi + 4*rdx + 28]  
31    add     rdx, 8  
32    cmp     rsi, rdx  
33    jne     .LBB0_9  
34  
35    .LBB0_4:  
36    test     rcx, rcx  
37    je      .LBB0_7  
38    lea     rdx, [rdi + 4*rdx]  
39    xor     esi, esi  
40  
41    .LBB0_6: # =>This Inner Loop Header: Depth=1  
42    add     eax, dword ptr [rdx + 4*rsi]  
43    inc     rsi  
44    cmp     rcx, rsi  
45    jne     .LBB0_6  
46  
47    .LBB0_7:  
48    ret
```

loop guard

extra prologue
(%, etc)

prologue

pl x8

iter x8
check + be

loop guard
prologue

iter
check + be

Examples. Loop (GCC vs CLANG)

The image displays two side-by-side assembly code snippets generated by GCC and Clang, comparing their output for a loop. The GCC side (left) shows a simple loop with a guard and a prologue. The Clang side (right) shows a more complex loop with a guard, prologue, and body. Annotations highlight differences like 'loop guard', 'extra prologue', 'prologue', 'p1 x8', 'iter x8 check + be', and 'loop guard prologue'.

Compiler Settings:

- Left: x86-64 gcc 12.2, -O2 -march=skylake -fno-tree-
- Right: x86-64 clang 15.0.0, -O2 -march=skylake -fno-vectorize -fno-

Assembly Code:

Left (GCC):

```
1 sum(int*, int):
2     test    esi, esi
3     jle     .L4
4     movsx   rsi, esi
5     lea     rdx, [rdi+rsi*4]
6     xor     eax, eax
7
8 .L3:
9     add     eax, DWORD PTR [rdi]
10    add     rdi, 4
11    cmp     rdi, rdx
12    jne     .L3
13
14 .L4:
15    xor     eax, eax
16    ret
```

Right (Clang):

```
1 sum(int*, int):                                # @sum(int*, int)
2     test    esi, esi
3     jle     .LBB0_1
4     mov     esi, esi
5     lea     rax, [rsi - 1]
6     mov     ecx, esi
7     and     ecx, 7
8     cmp     rax, 7
9     jae     .LBB0_8
10    xor     edx, edx
11    xor     eax, eax
12    jmp     .LBB0_4
13
14 .LBB0_1:
15    xor     eax, eax
16    ret
17
18 .LBB0_8:
19
20 .LBB0_9:                                # =>This Inner Loop Header: Depth=1
21    add     eax, dword ptr [rdi + 4*rdx]
22    add     eax, dword ptr [rdi + 4*rdx + 4]
23    add     eax, dword ptr [rdi + 4*rdx + 8]
24    add     eax, dword ptr [rdi + 4*rdx + 12]
25    add     eax, dword ptr [rdi + 4*rdx + 16]
26    add     eax, dword ptr [rdi + 4*rdx + 20]
27    add     eax, dword ptr [rdi + 4*rdx + 24]
28    add     eax, dword ptr [rdi + 4*rdx + 28]
29    add     rdx, 8
30    cmp     rsi, rdx
31    jne     .LBB0_9
32
33 .LBB0_4:
34    test     rcx, rcx
35    je      .LBB0_7
36    lea     rdx, [rdi + 4*rdx]
37    xor     esi, esi
38
39 .LBB0_6:                                # =>This Inner Loop Header: Depth=1
40    add     eax, dword ptr [rdx + 4*rsi]
41    inc     rsi
42    cmp     rcx, rsi
43    jne     .LBB0_6
44
45 .LBB0_7:
46    ret
```

Annotations:

- loop guard:** Lines 2-3 in Clang assembly.
- extra prologue (% , etc):** Lines 4-12 in Clang assembly.
- prologue:** Lines 17-19 in Clang assembly.
- p1 x8:** Lines 21-28 in Clang assembly.
- iter x8 check + be:** Lines 29-31 in Clang assembly.
- loop guard prologue:** Lines 33-36 in Clang assembly.

Examples. Loop (GCC vs CLANG)

The image compares the assembly output of GCC and Clang for a loop. The GCC output (left) shows a simple loop with a guard and a prologue. The Clang output (right) shows a more complex loop with multiple prologues and guards, indicating loop unrolling.

Where is unrolling ?!

loop guard

extra prologue (% , etc)

prologue

pl x8

iter x8 check + be

loop guard prologue

pl

iter check + be

GCC vs CLANG (-O2)

GCC:

- using PGO data or pragma

Clang:

- auto detect unroll and unroll count

GCC vs CLANG (-O2)

```
1 int sum(int *a, int n) {  
2     int x = 0;  
3     #pragma GCC unroll 4  
4     for (int i = 0; i < n; ++i)  
5         x += a[i];  
6     return x;  
7 }  
8
```

**force GCC to
unroll loop**

GCC vs CLANG (-O2)

```
x86-64 gcc 12.2  -O2 -march=skylake -fno-tree-vec

A  Output...  Filter...  Libraries  + Add new...  Add tool...

1  sum(int*, int):
2      test    esi, esi
3      jle     .L4
4      movsx   rsi, esi
5      lea     rdx, [-4+rsi*4]
6      shr     rdx, 2
7      inc     rdx
8      lea     rcx, [rdi+rsi*4]
9      xor     eax, eax
10     and     edx, 3
11     je      .L3
12     cmp     rdx, 1
13     je      .L15
14     cmp     rdx, 2
15     je      .L16
16     mov     eax, DWORD PTR [rdi]
17     add     rdi, 4
18 .L16:
19     add     eax, DWORD PTR [rdi]
20     add     rdi, 4
21 .L15:
22     add     eax, DWORD PTR [rdi]
23     add     rdi, 4
24     cmp     rdi, rcx
25     je      .L22
26 .L3:
27     add     eax, DWORD PTR [rdi]
28     add     eax, DWORD PTR [rdi+4]
29     add     eax, DWORD PTR [rdi+8]
30     add     eax, DWORD PTR [rdi+12]
31     add     rdi, 16
32     cmp     rdi, rcx
33     jne     .L3
34     ret
35 .L22:
36     ret
37 .L4:
38     xor     eax, eax
39     ret
```

GCC vs CLANG (-O2)

```
x86-64 gcc 12.2  -O2 -march=skylake -fno-tree-vec

1  sum(int*, int):
2      test    esi, esi
3      jle     .L4
4      movsx   rsi, esi
5      lea     rdx, [-4+rsi*4]
6      shr     rdx, 2
7      inc     rdx
8      lea     rcx, [rdi+rsi*4]
9      xor     eax, eax
10     and     edx, 3
11     je      .L3
12     cmp     rdx, 1
13     je      .L15
14     cmp     rdx, 2
15     je      .L16
16     mov     eax, DWORD PTR [rdi]
17     add     rdi, 4
18 .L16:
19     add     eax, DWORD PTR [rdi]
20     add     rdi, 4
21 .L15:
22     add     eax, DWORD PTR [rdi]
23     add     rdi, 4
24     cmp     rdi, rcx
25     je      .L22
26 .L3:
27     add     eax, DWORD PTR [rdi]
28     add     eax, DWORD PTR [rdi+4]
29     add     eax, DWORD PTR [rdi+8]
30     add     eax, DWORD PTR [rdi+12]
31     add     rdi, 16
32     cmp     rdi, rcx
33     jne     .L3
34     ret
35 .L22:
36     ret
37 .L4:
38     xor     eax, eax
39     ret
```

x4

GCC vs CLANG (-O2)

GCC:

- using PGO data or pragma
- trick with jumps and body cloning

Clang:

- auto detect unroll and unroll count
- “naive”

GCC vs CLANG (-O2)

```
int sum(int *a, int n) {  
    int x = 0;  
    #pragma GCC unroll 5  
    for (int i = 0; i < n; ++i)  
        x += a[i];  
    return x;  
}
```

force to
unroll 5

x86-64 gcc 12.2

A

```
14      cmp     rdx, 2  
15      je      .L16  
16      mov     eax, DWORD PTR [rdi]  
17      add     rdi, 4  
18  .L16:  
19      add     eax, DWORD PTR [rdi]  
20      add     rdi, 4  
21  .L15:  
22      add     eax, DWORD PTR [rdi]  
23      add     rdi, 4  
24      cmp     rdi, rcx  
25      je      .L22  
26  .L3:  
27      add     eax, DWORD PTR [rdi]  
28      add     eax, DWORD PTR [rdi+4]  
29      add     eax, DWORD PTR [rdi+8]  
30      add     eax, DWORD PTR [rdi+12]  
31      add     rdi, 16  
32      cmp     rdi, rcx  
33      jne     .L3  
34      ret  
35  .L22:  
36      ret  
37  .L4:  
38      xor     eax, eax  
39      ret
```

x4 !

x86-64 clang 15.0.0

A

Output...

Filter...

Libraries

+ Add new...

+ Add tool...

```
18      cmp     eax, 4  
19      jae     .LBB0_8  
20      xor     edx, edx  
21      xor     eax, eax  
22      jmp     .LBB0_4  
23  .LBB0_1:  
24      xor     eax, eax  
25      ret  
26  .LBB0_8:  
27      sub     esi, ecx  
28      xor     edx, edx  
29      xor     eax, eax  
30  .LBB0_9:                                     # =>This Inr  
31      add     eax, dword ptr [rdi + 4*rdx]  
32      add     eax, dword ptr [rdi + 4*rdx + 4]  
33      add     eax, dword ptr [rdi + 4*rdx + 8]  
34      add     eax, dword ptr [rdi + 4*rdx + 12]  
35      add     eax, dword ptr [rdi + 4*rdx + 16]  
36      add     rdx, 5  
37      cmp     esi, edx  
38      jne     .LBB0_9  
39  .LBB0_4:  
40      test    ecx, ecx  
41      je      .LBB0_7  
42      lea     rdx, [rdi + 4*rdx]  
43      mov     ecx, ecx  
44      xor     esi, esi  
45  .LBB0_6:                                     # =>This Inr  
46      add     eax, dword ptr [rdx + 4*rsi]  
47      inc     rsi
```

x5

GCC vs CLANG (-O2)

```
int sum(int *a, int n) {  
    int x = 0;  
    #pragma GCC unroll 5  
    for (int i = 0; i < n; ++i)  
        x += a[i];  
    return x;  
}
```

force to
unroll 5

No GCC
warning!

x86-64 gcc 12.2 -O2 -march=skylake -fno-
A
14 cmp rdx, 2
15 je .L16
16 mov eax, DWORD PTR [rdi]
17 add rdi, 4
18 .L16:
19 add eax, DWORD PTR [rdi]
20 add rdi, 4
21 .L15:
22 add eax, DWORD PTR [rdi]
23 add rdi, 4
24 cmp rdi, rcx
25 je .L22
26 .L3:
27 add eax, DWORD PTR [rdi]
28 add eax, DWORD PTR [rdi+4]
29 add eax, DWORD PTR [rdi+8]
30 add eax, DWORD PTR [rdi+12]
31 add rdi, 16
32 cmp rdi, rcx
33 jne .L3 x4 !
34 ret
35 .L22:
36 ret
37 .L4:
38 xor eax, eax
39 ret

x86-64 clang 15.0.0 -O2 -march=skylake -fno-vector
A Output... Filter... Libraries + Add new... + Add tool...
18 cmp eax, 4
19 jae .LBB0_8
20 xor edx, edx
21 xor eax, eax
22 jmp .LBB0_4
23 .LBB0_1:
24 xor eax, eax
25 ret
26 .LBB0_8:
27 sub esi, ecx
28 xor edx, edx
29 xor eax, eax
30 .LBB0_9: # =>This Inr
31 add eax, dword ptr [rdi + 4*rdx]
32 add eax, dword ptr [rdi + 4*rdx + 4]
33 add eax, dword ptr [rdi + 4*rdx + 8]
34 add eax, dword ptr [rdi + 4*rdx + 12]
35 add eax, dword ptr [rdi + 4*rdx + 16]
36 add rdx, 5
37 cmp esi, edx
38 jne .LBB0_9 x5
39 .LBB0_4:
40 test ecx, ecx
41 je .LBB0_7
42 lea rdx, [rdi + 4*rdx]
43 mov ecx, ecx
44 xor esi, esi
45 .LBB0_6: # =>This Inr
46 add eax, dword ptr [rdx + 4*rsi]
47 inc rsi

GCC vs CLANG (-O2)

```
A- [Icons] C++
1 int sum(int *a, int n) {
2     int x = 0;
3     #pragma GCC unroll 4
4     for (int i = 0; i < n; i += 3)
5         x += a[i];
6     return x;
7 }
8
```

non-power-of 2
loop step

```
x86-64 gcc 12.2 -O2 -march=skylake -fno-
A- [Icons]
1 sum(int*, int):
2     test    esi, esi
3     jle     .L4
4     xor     eax, eax
5     xor     edx, edx
6 .L3:
7     lea     rcx, [rax+3]
8     add     edx, DWORD PTR [rdi+rax*4]
9     cmp     esi, ecx
10    jle     .L1
11    lea     rax, [rcx+3]
12    add     edx, DWORD PTR [rdi+rcx*4]
13    cmp     esi, eax
14    jle     .L1
15    add     edx, DWORD PTR [rdi+rax*4]
16    lea     rax, [rcx+6]
17    cmp     esi, eax
18    jle     .L1
19    add     edx, DWORD PTR [rdi+rax*4]
20    lea     rax, [rcx+9]
21    cmp     esi, eax
22    jg      .L3
23 .L1:
24     mov     eax, edx
25     ret
26 .L4:
27     xor     edx, edx
28     mov     eax, edx
29     ret
```

```
x86-64 clang 15.0.0 -O2 -march=skylake -fno-vector
A- [Icons] Output... Filter... Libraries + Add new... Add tool...
16     ret
17 .LBB0_8:
18     mov     edx, r8d
19     and     edx, 2147483644
20     xor     esi, esi
21     xor     eax, eax
22 .LBB0_9:                                     # =>This Inr
23     add     eax, dword ptr [rdi + 4*rsi]
24     add     eax, dword ptr [rdi + 4*rsi + 12]
25     add     eax, dword ptr [rdi + 4*rsi + 24]
26     add     eax, dword ptr [rdi + 4*rsi + 36]
27     add     rsi, 12
28     add     edx, -4
29     jne     .LBB0_9
30 .LBB0_4:
31     test    r8b, 3
32     je      .LBB0_7
33     lea     rdx, [rdi + 4*rsi]
34     inc     cl
35     movzx   ecx, cl
36     and     ecx, 3
37     shl     rcx, 2
38     lea     rcx, [rcx + 2*rcx]
39     xor     esi, esi
40 .LBB0_6:                                     # =>This Inr
41     add     eax, dword ptr [rdx + rsi]
42     add     rsi, 12
43     cmp     ecx, esi
44     jne     .LBB0_6
45 .LBB0_7:
```

GCC vs CLANG (-O2)

```
A- + - v [icon] [icon] C++
1 int sum(int *a, int n) {
2     int x = 0;
3     #pragma GCC unroll 4
4     for (int i = 0; i < n; i += 3)
5         x += a[i];
6     return x;
7 }
8
```

non-power-of 2
loop step

```
x86-64 gcc 12.2 -O2 -march=skylake -fno-
A- [icon] [icon] [icon] [icon] [icon] [icon]
1 sum(int*, int):
2     test    esi, esi
3     jle     .L4
4     xor     eax, eax
5     xor     edx, edx
6 .L3:
7     lea     rcx, [rax+3]
8     add     edx, DWORD PTR [rdi+rax*4]
9     cmp     esi, ecx
10    jle     .L1
11    lea     rax, [rcx+3]
12    add     edx, DWORD PTR [rdi+rcx*4]
13    cmp     esi, eax
14    jle     .L1
15    add     edx, DWORD PTR [rdi+rax*4]
16    lea     rax, [rcx+6]
17    cmp     esi, eax
18    jle     .L1
19    add     edx, DWORD PTR [rdi+rax*4]
20    lea     rax, [rcx+9]
21    cmp     esi, eax
22    jg      .L3
23 .L1:
24    mov     eax, edx
25    ret
26 .L4:
27    xor     edx, edx
28    mov     eax, edx
29    ret
```

```
x86-64 clang 15.0.0 -O2 -march=skylake -fno-vector
A- [icon] [icon] [icon] [icon] [icon] [icon]
16     ret
17 .LBB0_8:
18     mov     edx, r8d
19     and     edx, 2147483644
20     xor     esi, esi
21     xor     eax, eax
22 .LBB0_9: # =>This Inr
23     add     eax, dword ptr [rdi + 4*rsi]
24     add     eax, dword ptr [rdi + 4*rsi + 12]
25     add     eax, dword ptr [rdi + 4*rsi + 24]
26     add     eax, dword ptr [rdi + 4*rsi + 36]
27     add     rsi, 12
28     add     edx, -4
29     jne     .LBB0_9
30 .LBB0_4:
31     test    r8b, 3
32     je      .LBB0_7
33     lea     rdx, [rdi + 4*rsi]
34     inc     cl
35     movzx   ecx, cl
36     and     ecx, 3
37     shl     rcx, 2
38     lea     rcx, [rcx + 2*rcx]
39     xor     esi, esi
40 .LBB0_6: # =>This Inr
41     add     eax, dword ptr [rdx + rsi]
42     add     rsi, 12
43     cmp     ecx, esi
44     jne     .LBB0_6
45 .LBB0_7:
```


GCC vs CLANG (-O2)

GCC:

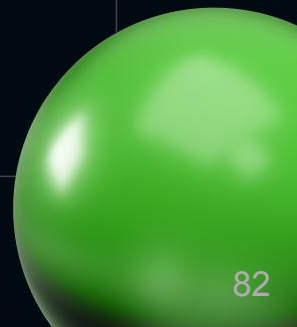
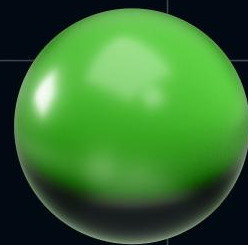
- using PGO data or pragma
- trick with jumps and body cloning
- Power-of-2 step && unroll count (rejection / body cloning)

Clang:

- auto detect unroll and unroll count
- “naive”
- any step / unroll count (prologue cost)

Outline of the talk

- Loop from compiler perspective
- Loop unrolling basics
- Loop unrolling overhead
- GCC and CLANG unroll details
- New optimization opportunities after unrolling
- Example



New opportunities

- Loop vectorization
- SLP vectorization
- Full unroll / loop deletion
- (Mem2Reg) Promote Memory To Register /
(SROA) Scalar Replacement Of Aggregates
- ...

Full unroll / loop deletion

```
1 int sum(int *a) {  
2     int x = 0;  
3     for (int i = 0; i < 4; i++)  
4         x += a[i];  
5     return x;  
6 }  
7
```

x86-64 gcc 12.2 -O2 -march=skylake -fno-tree-vectorize

```
1 sum(int*):  
2     mov     eax, DWORD PTR [rdi+4]  
3     add     eax, DWORD PTR [rdi]  
4     add     eax, DWORD PTR [rdi+8]  
5     add     eax, DWORD PTR [rdi+12]  
6     ret
```

x86-64 clang 15.0.0 -O2 -march=skylake -fno-slp-vectorize -fno-vectoriz

```
1 sum(int*): # @sum(int*)  
2     mov     eax, dword ptr [rdi + 4]  
3     add     eax, dword ptr [rdi]  
4     add     eax, dword ptr [rdi + 8]  
5     add     eax, dword ptr [rdi + 12]  
6     ret
```

Full unroll / loop deletion

The image displays a code editor with three panels. The left panel shows the C++ source code for a function `sum(int *a)`. The loop `for (int i = 0; i < 4; i++)` is highlighted with a red box. The right panel shows the assembly output for two different compilers: gcc 12.2 and clang 15.0.0. Both compilers have generated assembly code that fully unrolls the loop, resulting in a series of `add` instructions that sum the elements of the array `a` from index 0 to 3. The assembly code for gcc 12.2 is as follows:

```
1 sum(int*):  
2     mov     eax, DWORD PTR [rdi+4]  
3     add     eax, DWORD PTR [rdi]  
4     add     eax, DWORD PTR [rdi+8]  
5     add     eax, DWORD PTR [rdi+12]  
6     ret
```

The assembly code for clang 15.0.0 is as follows:

```
1 sum(int*):                                # @sum(int*)  
2     mov     eax, dword ptr [rdi + 4]  
3     add     eax, dword ptr [rdi]  
4     add     eax, dword ptr [rdi + 8]  
5     add     eax, dword ptr [rdi + 12]  
6     ret
```

Full unroll / loop deletion

The image displays a side-by-side comparison of C++ source code and its assembly output for two different compilers, illustrating the effect of the `-fno-tree-vectorize` flag on loop unrolling and deletion.

Left Panel (GCC 12.2):

- Source Code:** A C++ function `sum(int *a)` that initializes `x = 0` and iterates over an array `a` from index 0 to 3, adding each element to `x`. The `for` loop is highlighted with a red box. The text `compile time bounds` is written in red at the bottom.
- Assembly Output:** The assembly for `sum(int*)` shows four instructions in a loop: `mov eax, DWORD PTR [rdi+4]`, `add eax, DWORD PTR [rdi]`, `add eax, DWORD PTR [rdi+8]`, and `add eax, DWORD PTR [rdi+12]`. These instructions are enclosed in a green box, and the text `loop is deleted` is written in green below them.

Right Panel (Clang 15.0.0):

- Source Code:** The same C++ function `sum(int *a)` is shown.
- Assembly Output:** The assembly for `sum(int*)` shows four instructions in a loop: `mov eax, dword ptr [rdi + 4]`, `add eax, dword ptr [rdi]`, `add eax, dword ptr [rdi + 8]`, and `add eax, dword ptr [rdi + 12]`. These instructions are enclosed in a green box, and the text `loop is deleted` is written in green below them.

Full unroll / loop deletion

```
1 int sum(int *a) {  
2     int x = 0;  
3     for (int i = 0; i < 8; i++)  
4         x += a[i];  
5     return x;  
6 }  
7
```

compile time bounds

x86-64 gcc 12.2 -O2 -march=skylake -fno-tree-vectorize

```
1 sum(int*):  
2     lea     rdx, [rdi+32]  
3     xor     eax, eax  
4 .L2:  
5     add     eax, DWORD PTR [rdi]  
6     add     rdi, 4  
7     cmp     rdi, rdx  
8     jne     .L2  
9     ret
```

x86-64 clang 15.0.0 -O2 -march=skylake -fno-slp-vectorize -fno-vectoriz

```
1 sum(int*):  
2     mov     eax, dword ptr [rdi + 4]  
3     add     eax, dword ptr [rdi]  
4     add     eax, dword ptr [rdi + 8]  
5     add     eax, dword ptr [rdi + 12]  
6     add     eax, dword ptr [rdi + 16]  
7     add     eax, dword ptr [rdi + 20]  
8     add     eax, dword ptr [rdi + 24]  
9     add     eax, dword ptr [rdi + 28]  
10    ret
```

Full unroll / loop deletion

```
1 int sum(int *a) {  
2     int x = 0;  
3     for (int i = 0; i < 8; i++)  
4         x += a[i];  
5     return x;  
6 }  
7
```

compile time bounds

x86-64 gcc 12.2 -O2 -march=skylake -fno-tree-vectorize

```
1 sum(int*):  
2     lea     rdx, [rdi+32]  
3     xor     eax, eax  
4 .L2:  
5     add     eax, DWORD PTR [rdi]  
6     add     rdi, 4  
7     cmp     rdi, rdx  
8     jne     .L2  
9     ret
```

here is the loop
(no loop guard)

x86-64 clang 15.0.0 -O2 -march=skylake -fno-slp-vectorize -fno-vectoriz

```
1 sum(int*):  
2     mov     eax, dword ptr [rdi + 4]  
3     add     eax, dword ptr [rdi]  
4     add     eax, dword ptr [rdi + 8]  
5     add     eax, dword ptr [rdi + 12]  
6     add     eax, dword ptr [rdi + 16]  
7     add     eax, dword ptr [rdi + 20]  
8     add     eax, dword ptr [rdi + 24]  
9     add     eax, dword ptr [rdi + 28]  
10    ret
```

loop is deleted

Full unroll / loop deletion

```
1 int sum(int *a) {  
2     int x = 0;  
3     #pragma GCC unroll 8  
4     for (int i = 0; i < 8; i++)  
5         x += a[i];  
6     return x;  
7 }  
8
```

compile time bounds
+ pragma hint

x86-64 gcc 12.2 -O2 -march=skylake -fno-tree-vectorize

```
1 sum(int*):  
2     mov     eax, DWORD PTR [rdi+4]  
3     add     eax, DWORD PTR [rdi]  
4     add     eax, DWORD PTR [rdi+8]  
5     add     eax, DWORD PTR [rdi+12]  
6     add     eax, DWORD PTR [rdi+16]  
7     add     eax, DWORD PTR [rdi+20]  
8     add     eax, DWORD PTR [rdi+24]  
9     add     eax, DWORD PTR [rdi+28]  
10    ret
```

x86-64 clang 15.0.0 -O2 -march=skylake -fno-slp-vectorize -fno-vectoriz

```
1 sum(int*): # @sum(i  
2     mov     eax, dword ptr [rdi + 4]  
3     add     eax, dword ptr [rdi]  
4     add     eax, dword ptr [rdi + 8]  
5     add     eax, dword ptr [rdi + 12]  
6     add     eax, dword ptr [rdi + 16]  
7     add     eax, dword ptr [rdi + 20]  
8     add     eax, dword ptr [rdi + 24]  
9     add     eax, dword ptr [rdi + 28]  
10    ret
```

Full unroll / loop deletion

```
1 int sum(int *a) {  
2     int x = 0;  
3     for (int i = 0; i < 1024; i++)  
4         x += a[i];  
5     return x;  
6 }  
7
```

compile time bounds

x86-64 gcc 12.2 -O2 -march=skylake -fno-tree-vectorize

```
1 sum(int*):  
2     lea     rdx, [rdi+4096]  
3     xor     eax, eax  
4 .L2:  
5     add     eax, DWORD PTR [rdi]  
6     add     rdi, 4  
7     cmp     rdi, rdx  
8     jne     .L2  
9     ret
```

no unroll
no loop guard

x86-64 clang 15.0.0 -O2 -march=skylake -fno-slp-vectorize -fno-vectoriz

```
1 sum(int*):  
2     xor     ecx, ecx  
3     xor     eax, eax  
4 .LBB0_1:  
5     add     eax, dword ptr [rdi + 4*rcx]  
6     add     eax, dword ptr [rdi + 4*rcx + 4]  
7     add     eax, dword ptr [rdi + 4*rcx + 8]  
8     add     eax, dword ptr [rdi + 4*rcx + 12]  
9     add     eax, dword ptr [rdi + 4*rcx + 16]  
10    add     eax, dword ptr [rdi + 4*rcx + 20]  
11    add     eax, dword ptr [rdi + 4*rcx + 24]  
12    add     eax, dword ptr [rdi + 4*rcx + 28]  
13    add     eax, dword ptr [rdi + 4*rcx + 32]  
14    add     eax, dword ptr [rdi + 4*rcx + 36]  
15    add     eax, dword ptr [rdi + 4*rcx + 40]  
16    add     eax, dword ptr [rdi + 4*rcx + 44]  
17    add     eax, dword ptr [rdi + 4*rcx + 48]  
18    add     eax, dword ptr [rdi + 4*rcx + 52]  
19    add     eax, dword ptr [rdi + 4*rcx + 56]  
20    add     eax, dword ptr [rdi + 4*rcx + 60]  
21    add     rcx, 16  
22    cmp     rcx, 1024  
23    jne     .LBB0_1  
24    ret
```

unroll x16
no loop guard
no extra prologue

GCC vs CLANG (-O2)

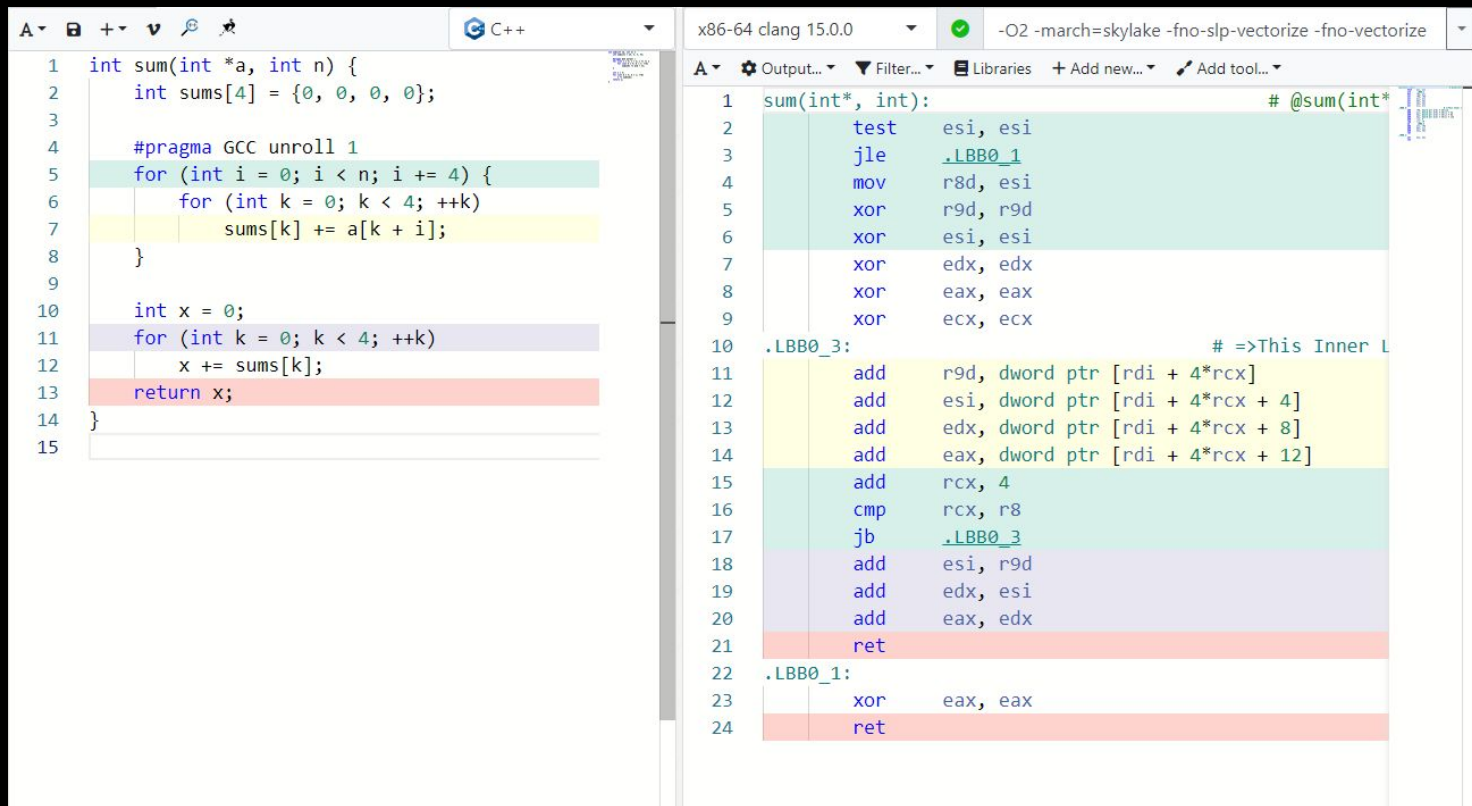
GCC:

- using PGO data or pragma
- trick with jumps and body cloning
- Power-of-2 step && unroll count (rejection / body cloning)
- Less aggressive full unroll

Clang:

- auto detect unroll and unroll count
- “naive”
- any step / unroll count (prologue cost)
- More aggressive full unroll

Mem2Reg / SROA (scalar replacement of aggregates)



The image shows a side-by-side comparison of C++ source code and its compiled assembly. The left pane displays the source code for a function `sum` that iterates over an array `sums` and calculates a result `x`. The right pane shows the corresponding assembly code generated by clang 15.0.0 with optimization flags `-O2 -march=skylake -fno-slp-vectorize -fno-vectorize`. The assembly includes labels like `.LBB0_1` and `.LBB0_3`, and instructions such as `test esi, esi`, `jle .LBB0_1`, `mov r8d, esi`, `xor r9d, r9d`, `xor esi, esi`, `xor edx, edx`, `xor eax, eax`, `xor ecx, ecx`, `add r9d, dword ptr [rdi + 4*rcx]`, `add esi, dword ptr [rdi + 4*rcx + 4]`, `add edx, dword ptr [rdi + 4*rcx + 8]`, `add eax, dword ptr [rdi + 4*rcx + 12]`, `add rcx, 4`, `cmp rcx, r8`, `jb .LBB0_3`, `add esi, r9d`, `add edx, esi`, `add eax, edx`, `ret`, and `xor eax, eax`.

```
1  int sum(int *a, int n) {  
2      int sums[4] = {0, 0, 0, 0};  
3  
4      #pragma GCC unroll 1  
5      for (int i = 0; i < n; i += 4) {  
6          for (int k = 0; k < 4; ++k)  
7              sums[k] += a[k + i];  
8      }  
9  
10     int x = 0;  
11     for (int k = 0; k < 4; ++k)  
12         x += sums[k];  
13     return x;  
14 }  
15
```

x86-64 clang 15.0.0 -O2 -march=skylake -fno-slp-vectorize -fno-vectorize

```
1  sum(int*, int):                                # @sum(int*, int)  
2      test     esi, esi  
3      jle     .LBB0_1  
4      mov     r8d, esi  
5      xor     r9d, r9d  
6      xor     esi, esi  
7      xor     edx, edx  
8      xor     eax, eax  
9      xor     ecx, ecx  
10     .LBB0_3:                                    # =>This Inner Loop Is Not Unrolled  
11         add     r9d, dword ptr [rdi + 4*rcx]  
12         add     esi, dword ptr [rdi + 4*rcx + 4]  
13         add     edx, dword ptr [rdi + 4*rcx + 8]  
14         add     eax, dword ptr [rdi + 4*rcx + 12]  
15         add     rcx, 4  
16         cmp     rcx, r8  
17         jb     .LBB0_3  
18         add     esi, r9d  
19         add     edx, esi  
20         add     eax, edx  
21         ret  
22     .LBB0_1:  
23         xor     eax, eax  
24         ret
```

Mem2Reg / SROA (scalar replacement of aggregates)

The image displays a C++ compiler's source and assembly views. The source code on the left is for a function `sum` that takes an array `a` and a size `n`, and returns the sum of elements at indices `k` and `k+i` for `k` from 0 to `n-4`. A red box highlights the line `sums[k] += a[k + i];` with a red arrow pointing to the text "write to mem (stack)". The assembly code on the right shows the compiled instructions, with corresponding blocks highlighted in yellow and purple.

Source Code (C++):

```
1 int sum(int *a, int n) {  
2     int sums[4] = {0, 0, 0, 0};  
3  
4     #pragma GCC unroll 1  
5     for (int i = 0; i < n; i += 4) {  
6         for (int k = 0; k < 4; ++k)  
7             sums[k] += a[k + i];  
8     }  
9  
10    int x = 0;  
11    for (int k = 0; k < 4; ++k)  
12        x += sums[k];  
13    return x;  
14 }  
15
```

Assembly Code (x86-64 clang 15.0.0):

```
1 sum(int*, int):                                # @sum(int*, int)  
2     test     esi, esi  
3     jle     .LBB0_1  
4     mov     r8d, esi  
5     xor     r9d, r9d  
6     xor     esi, esi  
7     xor     edx, edx  
8     xor     eax, eax  
9     xor     ecx, ecx  
10    .LBB0_3:                                    # =>This Inner Loop  
11        add     r9d, dword ptr [rdi + 4*rcx]  
12        add     esi, dword ptr [rdi + 4*rcx + 4]  
13        add     edx, dword ptr [rdi + 4*rcx + 8]  
14        add     eax, dword ptr [rdi + 4*rcx + 12]  
15        add     rcx, 4  
16        cmp     rcx, r8  
17        jbe     .LBB0_3  
18        add     esi, r9d  
19        add     edx, esi  
20        add     eax, edx  
21        ret  
22    .LBB0_1:  
23        xor     eax, eax  
24        ret
```

Mem2Reg / SROA (scalar replacement of aggregates)

Source Code (C++):

```
1 int sum(int *a, int n) {  
2     int sums[4] = {0, 0, 0, 0};  
3  
4     #pragma GCC unroll 1  
5     for (int i = 0; i < n; i += 4) {  
6         for (int k = 0; k < 4; ++k)  
7             sums[k] += a[k + i];  
8     }  
9  
10    int x = 0; write to mem (stack)  
11    for (int k = 0; k < 4; ++k)  
12        x += sums[k];  
13    return x;  
14 }  
15
```

Assembly Code (x86-64 clang 15.0.0):

```
1 sum(int*, int):                                # @sum(int*)  
2     test     esi, esi  
3     jle      .LBB0_1  
4     mov      r8d, esi  
5     xor      r9d, r9d  
6     xor      esi, esi  
7     xor      edx, edx read  
8     xor      eax, eax  
9     xor      ecx, ecx  
10    .LBB0_3:                                     # =>This Inner L  
11        add     r9d, dword ptr [rdi + 4*rcx]  
12        add     esi, dword ptr [rdi + 4*rcx + 4]  
13        add     edx, dword ptr [rdi + 4*rcx + 8]  
14        add     eax, dword ptr [rdi + 4*rcx + 12]  
15        add     rcx, 4  
16        cmp     rcx, r8  
17        jnb     .LBB0_3  
18        add     esi, r9d  
19        add     edx, esi  
20        add     eax, edx  
21        ret  
22    .LBB0_1:  
23        xor     eax, eax  
24        ret
```

Mem2Reg / SROA (scalar replacement of aggregates)

The image displays a C++ compiler's source code and its corresponding assembly output, illustrating the Mem2Reg / SROA (scalar replacement of aggregates) optimization.

Source Code (Left Pane):

```
1 int sum(int *a, int n) {  
2     int sums[4] = {0, 0, 0, 0};  
3  
4     #pragma GCC unroll 1  
5     for (int i = 0; i < n; i += 4) {  
6         for (int k = 0; k < 4; ++k)  
7             sums[k] += a[k + i];  
8     }  
9  
10    int x = 0; write to mem (stack)  
11    for (int k = 0; k < 4; ++k)  
12        x += sums[k];  
13    return x;  
14 }  
15
```

Assembly Output (Right Pane):

The assembly code is generated for x86-64 using clang 15.0.0, with optimization flags: `-O2 -march=skylake -fno-slp-vectorize -fno-vectorize`.

The assembly shows the function `sum(int*, int):` with the following instructions:

- `test esi, esi`
- `jle .LBB0_1`
- `mov r8d, esi`
- `xor r9d, r9d`
- `xor esi, esi`
- `xor edx, edx`
- `xor eax, eax`
- `xor ecx, ecx`
- `.LBB0_3:`
- `add r9d, dword ptr [rdi + 4*rcx]`
- `add esi, dword ptr [rdi + 4*rcx + 4]`
- `add edx, dword ptr [rdi + 4*rcx + 8]`
- `add eax, dword ptr [rdi + 4*rcx + 12]`
- `add rcx, 4`
- `cmp rcx, r8`
- `jb .LBB0_3`
- `add esi, r9d`
- `add edx, esi`
- `add eax, edx`
- `ret`
- `.LBB0_1:`
- `xor eax, eax`
- `ret`

Annotations in the assembly output highlight memory access patterns:

- `read` (green text) is placed next to `xor esi, esi` and `xor ecx, ecx`.
- `read` (green text) is placed next to `add esi, r9d`.
- `write to mem (stack)` (red text) is placed next to `add r9d, dword ptr [rdi + 4*rcx]`.

Mem2Reg / SROA (scalar replacement of aggregates)

The image displays a C++ compiler's source code and its corresponding assembly output, illustrating the Mem2Reg / SROA (scalar replacement of aggregates) optimization.

Source Code (Left Panel):

```
1 int sum(int *a, int n) {  
2     int sums[4] = {0, 0, 0, 0};  
3  
4     #pragma GCC unroll 1  
5     for (int i = 0; i < n; i += 4) {  
6         for (int k = 0; k < 4; ++k)  
7             sums[k] += a[k + i];  
8     }  
9  
10    int x = 0; write to mem (stack)  
11    for (int k = 0; k < 4; ++k)  
12        x += sums[k];  
13    return x;  
14 }  
15
```

Assembly Output (Right Panel):

The assembly output shows the compiled code for the function `sum(int*, int):`. The code is divided into blocks labeled `.LBB0_1` and `.LBB0_3`.

Block .LBB0_1:

```
2     test     esi, esi  
3     jle     .LBB0_1  
4     mov     r8d, esi  
5     xor     r9d, r9d  
6     xor     esi, esi  
7     xor     edx, edx  
8     xor     eax, eax  
9     xor     ecx, ecx
```

Block .LBB0_3:

```
10    .LBB0_3:  
11    add     r9d, dword ptr [rdi + 4*rcx]  
12    add     esi, dword ptr [rdi + 4*rcx + 4]  
13    add     edx, dword ptr [rdi + 4*rcx + 8]  
14    add     eax, dword ptr [rdi + 4*rcx + 12]  
15    add     rcx, 4  
16    cmp     rcx, r8  
17    jb     .LBB0_3  
18    add     esi, r9d  
19    add     edx, esi  
20    add     eax, edx  
21    ret
```

Block .LBB0_1:

```
22    .LBB0_1:  
23    xor     eax, eax  
24    ret
```

The assembly output includes annotations such as `read` and `write to mem (stack)` to highlight specific operations. Green boxes highlight memory address calculations in the assembly, and green text 'read' indicates memory reads.

Mem2Reg / SROA (scalar replacement of aggregates)

The image displays a C++ source code editor and its corresponding assembly output, illustrating the Mem2Reg / SROA (scalar replacement of aggregates) optimization.

Source Code (Left Pane):

```
1 int sum(int *a, int n) {  
2     int sums[4] = {0, 0, 0, 0};  
3  
4     #pragma GCC unroll 1  
5     for (int i = 0; i < n; i += 4) {  
6         for (int k = 0; k < 4; ++k)  
7             sums[k] += a[k + i];  
8     }  
9     int x = 0; write to mem (stack)  
10    for (int k = 0; k < 4; ++k)  
11        x += sums[k];  
12    return x;  
13 }  
14  
15
```

Assembly Output (Right Pane):

```
1 sum(int*, int): # @sum(int*)  
2     test     esi, esi  
3     jle      .LBB0_1  
4     mov      r8d, esi  
5     xor      r9d, r9d  
6     xor      esi, esi  
7     xor      edx, edx read  
8     xor      eax, eax read  
9     xor      ecx, ecx  
10    .LBB0_3: # =>This Inner L  
11        add     r9d, dword ptr [rdi + 4*rcx]  
12        add     esi, dword ptr [rdi + 4*rcx + 4]  
13        add     edx, dword ptr [rdi + 4*rcx + 8]  
14        add     eax, dword ptr [rdi + 4*rcx + 12]  
15        add     rcx, 4 read  
16        cmp     rcx, r8 read  
17        jb      .LBB0_3  
18        add     esi, r9d  
19        add     edx, esi  
20        add     eax, edx  
21        ret  
22    .LBB0_1:  
23        xor     eax, eax  
24        ret
```

Mem2Reg / SROA (scalar replacement of aggregates)

The image displays a side-by-side comparison of C++ source code and its compiled assembly output, illustrating the Mem2Reg / SROA optimization.

Left Panel (C++ Source Code):

```
1 int sum(int *a, int n) {  
2     int sums[4] = {0, 0, 0, 0};  
3  
4     #pragma GCC unroll 1  
5     for (int i = 0; i < n; i += 4) {  
6         for (int k = 0; k < 4; ++k)  
7             sums[k] += a[k + i];  
8     }  
9     int x = 0; write to mem (stack)  
10    for (int k = 0; k < 4; ++k)  
11        x += sums[k];  
12    return x;  
13 }  
14  
15
```

Right Panel (x86-64 Assembly Output):

Compiler: x86-64 clang 15.0.0
Options: -O2 -march=skylake -fno-slp-vectorize -fno-vectorize

```
1 sum(int*, int):                                # @sum(int*, int)  
2     test     esi, esi  
3     jle     .LBB0_1  
4     mov     r8d, esi  
5     xor     r9d, r9d  
6     xor     esi, esi  
7     xor     edx, edx read  
8     xor     eax, eax read  
9     xor     ecx, ecx  
10    .LBB0_3:                                     # =>This Inner Loop  
11        add     r9d, dword ptr [rdi + 4*rcx]  
12        add     esi, dword ptr [rdi + 4*rcx + 4]  
13        add     edx, dword ptr [rdi + 4*rcx + 8]  
14        add     eax, dword ptr [rdi + 4*rcx + 12]  
15        add     rcx, 4 read  
16        cmp     rcx, r8 read  
17        jbe     .LBB0_3 read  
18        add     esi, r9d  
19        add     edx, esi no writes!  
20        add     eax, edx  
21        ret  
22    .LBB0_1:  
23        xor     eax, eax  
24        ret
```

Mem2Reg / SROA (scalar replacement of aggregates)

```
int sum(int *a, int n) {  
    int sums[4] = {0, 0, 0, 0};  
  
    for (int i = 0; i < n; i += 4) {  
        for (int k = 0; k < 4; ++k)  
            sums[k] += a[k + i];  
    }  
  
    int x = 0;  
    for (int k = 0; k < 4; ++k)  
        x += sums[k];  
    return x;  
}
```

Mem2Reg / SROA (scalar replacement of aggregates)

```
int sum(int *a, int n) {  
    int sums[4] = {0, 0, 0, 0};  
  
    for (int i = 0; i < n; i += 4) {  
        for (int k = 0; k < 4; ++k)  
            sums[k] += a[k + i];  
    }  
  
    int x = 0;  
    for (int k = 0; k < 4; ++k)  
        x += sums[k];  
    return x;  
}
```

Mem2Reg / SROA (scalar replacement of aggregates)

```
int sum(int *a, int n) {  
    int sums[4] = {0, 0, 0, 0};  
  
    for (int i = 0; i < n; i += 4) {  
        for (int k = 0; k < 4; k += 1) {  
            sums[0] += a[i];  
            sums[1] += a[i + 1];  
            sums[2] += a[i + 2];  
            sums[3] += a[i + 3];  
        }  
    }  
  
    int x = 0;  
    for (int k = 0; k < 4; k += 1) {  
        x += sums[0];  
        x += sums[1];  
        x += sums[2];  
        x += sums[3];  
    }  
    return x;  
}
```

Mem2Reg / SROA (scalar replacement of aggregates)

```
int sum(int *a, int n) {  
    int sums[4] = {0, 0, 0, 0};  
  
    for (int i = 0; i < n; i += 4) {  
        for (int k = 0; k < 4; k += 1) {  
            sums[0] += a[i];  
            sums[1] += a[i + 1];  
            sums[2] += a[i + 2];  
            sums[3] += a[i + 3];  
        }  
    }  
  
    int x = 0;  
    for (int k = 0; k < 4; k += 1) {  
        x += sums[0];  
        x += sums[1];  
        x += sums[2];  
        x += sums[3];  
    }  
    return x;  
}
```

Mem2Reg / SROA (scalar replacement of aggregates)

```
int sum(int *a, int n) {  
    int sums[4] = {0, 0, 0, 0};  
  
    for (int i = 0; i < n; i += 4) {  
        sums[0] += a[i];  
        sums[1] += a[i + 1];  
        sums[2] += a[i + 2];  
        sums[3] += a[i + 3];  
    }  
  
    int x = 0;  
    x += sums[0];  
    x += sums[1];  
    x += sums[2];  
    x += sums[3];  
    return x;  
}
```

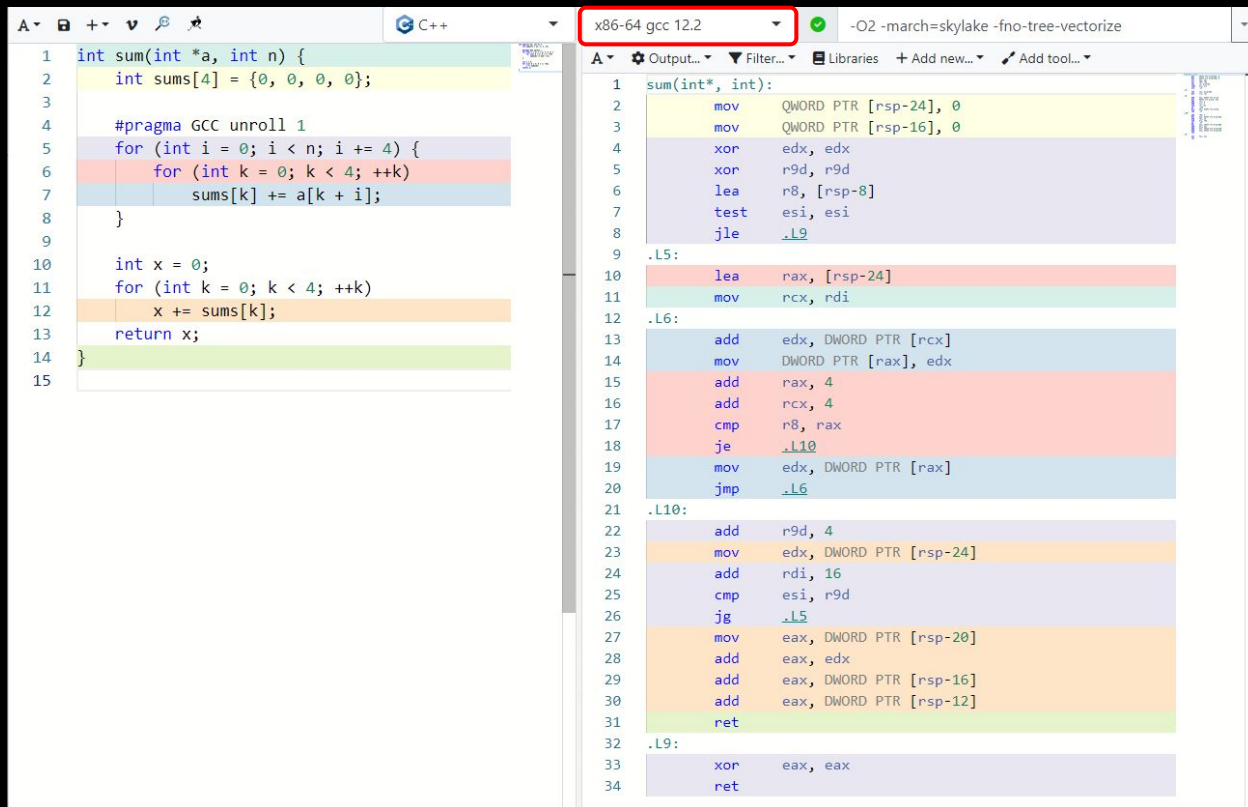
Mem2Reg / SROA (scalar replacement of aggregates)

```
int sum(int *a, int n) {  
    int sums[4] = {0, 0, 0, 0};  
  
    for (int i = 0; i < n; i += 4) {  
        sums[0] += a[i];  
        sums[1] += a[i + 1];  
        sums[2] += a[i + 2];  
        sums[3] += a[i + 3];  
    }  
  
    int x = 0;  
    x += sums[0];  
    x += sums[1];  
    x += sums[2];  
    x += sums[3];  
    return x;  
}
```


Mem2Reg / SROA (scalar replacement of aggregates)

```
int sum(int *a, int n) {  
    int sums_0 = 0;  
    int sums_1 = 0;  
    int sums_2 = 0;  
    int sums_3 = 0;  
  
    for (int i = 0; i < n; i += 4) {  
        sums_0 += a[i];  
        sums_1 += a[i + 1];  
        sums_2 += a[i + 2];  
        sums_3 += a[i + 3];  
    }  
  
    int x = 0;  
    x += sums_0;  
    x += sums_1;  
    x += sums_2;  
    x += sums_3;  
    return x;  
}
```

Mem2Reg / SROA (scalar replacement of aggregates)



```
1  int sum(int *a, int n) {
2      int sums[4] = {0, 0, 0, 0};
3
4      #pragma GCC unroll 1
5      for (int i = 0; i < n; i += 4) {
6          for (int k = 0; k < 4; ++k)
7              sums[k] += a[k + i];
8      }
9
10     int x = 0;
11     for (int k = 0; k < 4; ++k)
12         x += sums[k];
13     return x;
14 }
15
```

x86-64 gcc 12.2 -O2 -march=skylake -fno-tree-vectorize

```
1  sum(int*, int):
2      mov     QWORD PTR [rsp-24], 0
3      mov     QWORD PTR [rsp-16], 0
4      xor     edx, edx
5      xor     r9d, r9d
6      lea     r8, [rsp+8]
7      test    esi, esi
8      jle     .L9
9
10 .L5:
11     lea     rax, [rsp-24]
12     mov     rcx, rdi
13
14 .L6:
15     add     edx, DWORD PTR [rcx]
16     mov     DWORD PTR [rax], edx
17     add     rax, 4
18     add     rcx, 4
19     cmp     r8, rax
20     je      .L10
21     mov     edx, DWORD PTR [rax]
22     jmp     .L6
23
24 .L10:
25     add     r9d, 4
26     mov     edx, DWORD PTR [rsp-24]
27     add     rdi, 16
28     cmp     esi, r9d
29     jg      .L5
30     mov     eax, DWORD PTR [rsp-20]
31     add     eax, edx
32     add     eax, DWORD PTR [rsp-16]
33     add     eax, DWORD PTR [rsp-12]
34     ret
35
36 .L9:
37     xor     eax, eax
38     ret
39
```

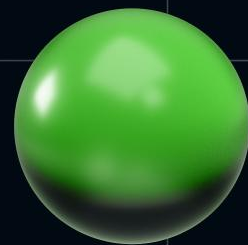
Mem2Reg / SROA (scalar replacement of aggregates)

```
1 int sum(int *a, int n) {
2     int sums[4] = {0, 0, 0, 0};
3
4     #pragma GCC unroll 1
5     for (int i = 0; i < n; i += 4) {
6         for (int k = 0; k < 4; ++k)
7             sums[k] += a[k + i];
8     }
9
10    int x = 0;
11    for (int k = 0; k < 4; ++k)
12        x += sums[k];
13    return x;
14 }
15
```

```
1 sum(int*, int):
2     mov     QWORD PTR [rsp-24], 0
3     mov     QWORD PTR [rsp-16], 0
4     xor     edx, edx
5     xor     r9d, r9d
6     lea     r8, [rsp-8]
7     test    esi, esi
8     jle     .L9
9
10 .L5:
11     lea     rax, [rsp-24]
12     mov     rcx, rdi
13
14 .L6:
15     add     edx, DWORD PTR [rcx]
16     mov     DWORD PTR [rax], edx
17     add     rax, 4
18     add     rcx, 4
19     cmp     r8, rax
20     je      .L10
21     mov     edx, DWORD PTR [rax]
22     jmp     .L6
23
24 .L10:
25     add     r9d, 4
26     mov     edx, DWORD PTR [rsp-24]
27     add     rdi, 16
28     cmp     esi, r9d
29     jg      .L5
30     mov     eax, DWORD PTR [rsp-20]
31     add     eax, edx
32     add     eax, DWORD PTR [rsp-16]
33     add     eax, DWORD PTR [rsp-12]
34     ret
35
36 .L9:
37     xor     eax, eax
38     ret
```

Outline of the talk

- Loop from compiler perspective
- Loop unrolling basics
- Loop unrolling overhead
- GCC and CLANG unroll details
- New optimization opportunities after unrolling
- Example

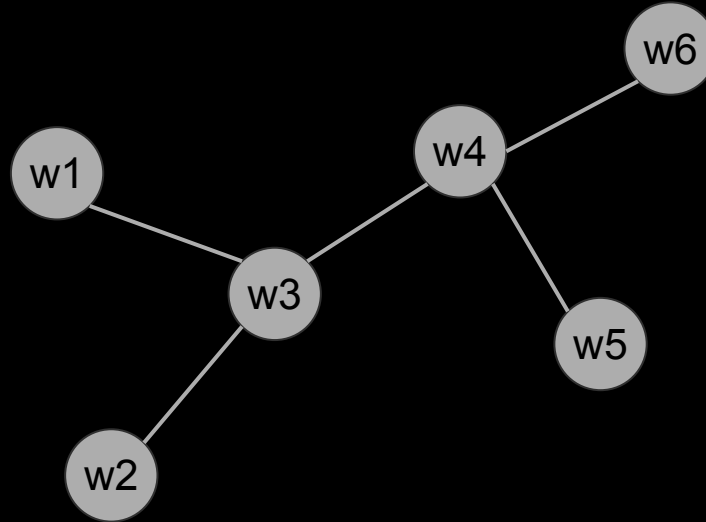


Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP

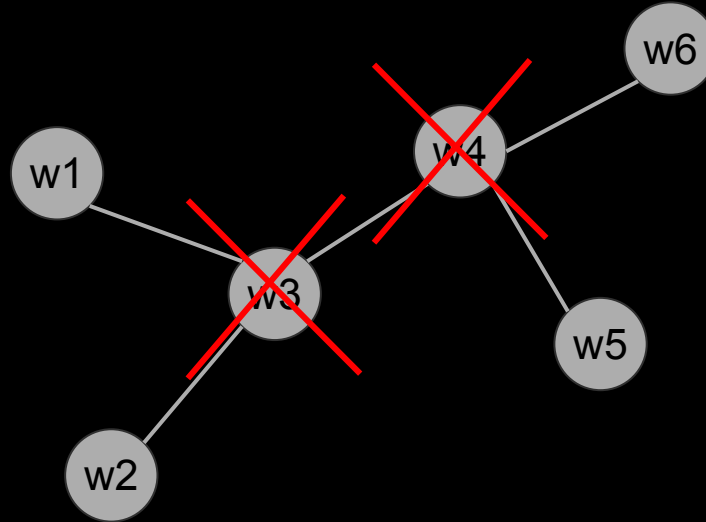
Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP



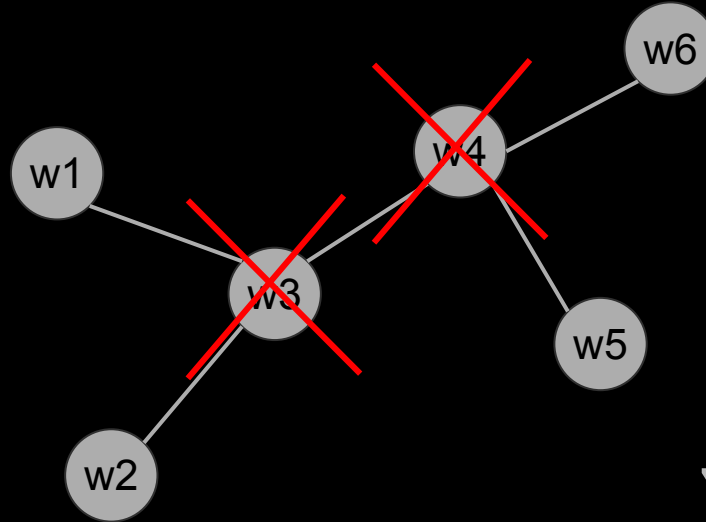
Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP



Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP

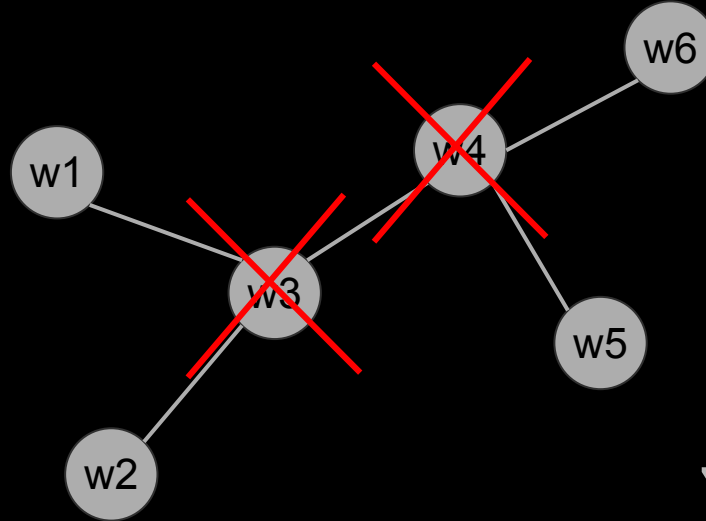


$$\sum_{\text{deleted}} w_i \rightarrow \min$$

Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP

NP-complete!



$$\sum_{\text{deleted}} w_i \rightarrow \min$$

Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP

But...

Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP

But...

- Assume weights $\in \mathbb{N}$

Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP

But...

- Assume weights $\in \mathbb{N}$
- Assume NumVertex ≤ 16

Example

Need to solve Minimum Weighted Vertex Cover Problem ASAP

But...

- Assume weights $\in \mathbb{N}$
- Assume NumVertex ≤ 16
- Encode task as... bits:
 - vertices set is given as bitmask (uint32_t)

Example

```
uint32_t getBestSubset(  
    const vector<uint32_t> &InW,  
    const vector<uint32_t> &InM) {  
    array<uint32_t, 16> Weights = {0};  
    array<uint32_t, 16> AdjMasks = {0};  
    copy(begin(InW), end(InW), begin(Weights));  
    copy(begin(InM), end(InM), begin(AdjMasks));  
}
```

Example

```
const auto GetSubsetWeight = [&](uint32_t Subset) {  
    uint32_t RV = 0;  
    for (size_t I = 0; I < 16; ++I)  
        RV += (Subset & (1 << I)) ? Weights[I] : 0;  
    return RV;  
};  
  
const auto IsCompatible = [&](uint32_t Subset) {  
    uint32_t IncompatibleVerMask = 0;  
    for (size_t I = 0; I < 16; ++I)  
        IncompatibleVerMask |= (Subset & (1 << I)) ? AdjMasks[I] : 0;  
    return (Subset & IncompatibleVerMask) == 0;  
};
```

Example

```
uint32_t BestWeight = 0;
uint32_t BestSubset = 0;
const uint32_t NumSubsets = 1 << InW.size();
for (uint32_t Subset = 1; Subset < NumSubsets; ++Subset) {

    const uint32_t Weight = GetSubsetWeight(Subset);
    if (Weight <= BestWeight)
        continue;

    if (!IsCompatible(Subset))
        continue;

    BestWeight = Weight;
    BestSubset = Subset;
}
return BestSubset;
}
```


Example

The image displays a C++ source code file on the left and its compiled assembly output on the right. The C++ code defines a function to find the best subset of weights based on compatibility and a subset mask. The assembly output shows the low-level instructions generated by the compiler, including vector operations and control flow.

```
17     const uint32_t NumVtx) {
18     array<uint32_t, N> Weights = {0};
19     array<uint32_t, N> AdjMasks = {0};
20     copy(begin(InW), end(InW), begin(Weights));
21     copy(begin(InM), end(InM), begin(AdjMasks));
22
23     const auto GetSubsetWeight = [&](uint32_t Subset) {
24         uint32_t RV = 0;
25         for (size_t I = 0; I < N; ++I)
26             RV += (Subset & (1 << I)) ? Weights[I] : 0;
27         return RV;
28     };
29
30     const auto IsCompatible = [&](uint32_t Subset) {
31         uint32_t IncompatMask = 0;
32         for (size_t I = 0; I < N; ++I)
33             IncompatMask |= (Subset & (1 << I)) ? AdjMasks[I] : 0;
34         return (Subset & IncompatMask) == 0;
35     };
36
37     uint32_t BestWeight = 0;
38     uint32_t BestSubset = 0;
39     const uint32_t NumSubsets = 1 << NumVtx;
40     for (uint32_t Subset = 1; Subset < NumSubsets; ++Subset) {
41         const uint32_t Weight = GetSubsetWeight(Subset);
42         if (Weight <= BestWeight)
43             continue;
44
45         if (!IsCompatible(Subset))
46             continue;
47
48         BestWeight = Weight;
49         BestSubset = Subset;
50     }
51     return BestSubset;
52 }
```

The assembly output (x86-64 clang 15.0.0, -O2 -march=skylake) shows the following instructions:

```
59     vpxor    xmm6, xmm6, xmm6
60     xor      eax, eax
61     jmp      .LBB0_8
62 .LBB0_10:                                     # in Loop: Header=BB0_8 Depth=1
63     inc     ecx
64     shr     esi, ecx, ebx
65     test    esi, esi
66     jne     .LBB0_6
67 .LBB0_8:                                     # =>This Inner Loop Header: Depth=1
68     vmovd   xmm7, ecx
69     vpbroadcastq ymm7, xmm7
70     vpand   ymm8, ymm7, ymm4
71     vpand   ymm7, ymm7, ymm5
72     vpcmpq   ymm9, ymm7, ymm5
73     vpand   ymm9, ymm9, ymm1
74     vpcmpq   ymm10, ymm8, ymm4
75     vpand   ymm10, ymm10, ymm0
76     vpaddq   ymm9, ymm10, ymm9
77     vextracti128 xmm2, ymm9, 1
78     vpaddq   xmm2, xmm9, xmm2
79     vpslufd xmm3, xmm2, 238                  # xmm3 = xmm2[2,3,2,3]
80     vpaddq   xmm2, xmm2, xmm3
81     vpslufd xmm3, xmm2, 85                   # xmm3 = xmm2[1,1,1,1]
82     vpaddq   xmm2, xmm2, xmm3
83     vmovd   esi, xmm2
84     cmp     esi, edx
85     jbe     .LBB0_10
86     vpcmpgtd ymm2, ymm8, ymm6
87     vpand   ymm2, ymm11, ymm2
88     vpcmpgtd ymm3, ymm7, ymm6
89     vpand   ymm3, ymm12, ymm3
90     vpor    ymm2, ymm2, ymm3
91     vextracti128 xmm3, ymm2, 1
92     vpor    xmm2, xmm2, xmm3
93     vpslufd xmm3, xmm2, 238                  # xmm3 = xmm2[2,3,2,3]
94     vpor    xmm2, xmm2, xmm3
95     vpslufd xmm3, xmm2, 85                   # xmm3 = xmm2[1,1,1,1]
96     vpor    xmm2, xmm2, xmm3
97     vmovd   edi, xmm2
98     test    edi, ecx
99     cmov     eax, ecx
100    cmov     edx, esi
101    jmp      .LBB0_10
```

Example

```
17 const uint32_t NumVtx {
18 array<uint32_t, N> Weights = {0};
19 array<uint32_t, N> AdjMasks = {0};
20 copy(begin(InW), end(InW), begin(Weights));
21 copy(begin(InM), end(InM), begin(AdjMasks));
22
23 const auto GetSubsetWeight = [&](uint32_t Subset) {
24     uint32_t RV = 0;
25     for (size_t I = 0; I < N; ++I)
26         RV += (Subset & (1 << I)) ? Weights[I] : 0;
27     return RV;
28 };
29
30 const auto IsCompatible = [&](uint32_t Subset) {
31     uint32_t IncompatMask = 0;
32     for (size_t I = 0; I < N; ++I)
33         IncompatMask |= (Subset & (1 << I)) ? AdjMasks[I] : 0;
34     return (Subset & IncompatMask) == 0;
35 };
36
37 uint32_t BestWeight = 0;
38 uint32_t BestSubset = 0;
39 const uint32_t NumSubsets = 1 << NumVtx;
40 for (uint32_t Subset = 1; Subset < NumSubsets; ++Subset) {
41     const uint32_t Weight = GetSubsetWeight(Subset);
42     if (Weight <= BestWeight)
43         continue;
44     if (!IsCompatible(Subset))
45         continue;
46     BestWeight = Weight;
47     BestSubset = Subset;
48 }
49 return BestSubset;
```

```
59 vpxor xmm6, xmm6, xmm6
60 xor eax, eax
61 jmp .LBB0_8
62 .LBB0_10: # in Loop: Header=BB0_8 Depth=1
63 inc ecx
64 shrq esi, ecx, ebx
65 test esi, esi
66 jne .LBB0_6
67 .LBB0_8: # =>This Inner Loop Header: Depth=1
68 vmovd xmm7, ecx
69 vpbroadcastq ymm7, xmm7
70 vpand ymm8, ymm7, ymm4
71 vpand ymm7, ymm7, ymm5
72 vpcmpq ymm9, ymm7, ymm5
73 vpand ymm9, ymm9, ymm1
74 vpcmpq ymm10, ymm8, ymm4
75 vpand ymm10, ymm10, ymm0
76 vpaddq ymm9, ymm10, ymm9
77 vextracti128 xmm2, ymm9, 1
78 vpaddq xmm2, xmm9, xmm2
79 vpslufd xmm3, xmm2, 238 # xmm3 = xmm2[2,3,2,3]
80 vpaddq xmm2, xmm2, xmm3
81 vpslufd xmm3, xmm2, 85 # xmm3 = xmm2[1,1,1,1]
82 vpaddq xmm2, xmm2, xmm3
83 vmovd esi, xmm2
84 cmp esi, edx
85 jbe .LBB0_10
86 vpcmpgtd ymm2, ymm8, ymm6
87 vpand ymm2, ymm11, ymm2
88 vpcmpgtd ymm3, ymm7, ymm6
89 vpand ymm3, ymm12, ymm3
90 vpor ymm2, ymm2, ymm3
91 vextracti128 xmm3, ymm2, 1
92 vpor xmm2, xmm2, xmm3
93 vpslufd xmm3, xmm2, 238 # xmm3 = xmm2[2,3,2,3]
94 vpor xmm2, xmm2, xmm3
95 vpslufd xmm3, xmm2, 85 # xmm3 = xmm2[1,1,1,1]
96 vpor xmm2, xmm2, xmm3
97 vmovd edi, xmm2
98 test edi, ecx
99 cmovle eax, ecx
100 cmovle edx, esi
101 jmp .LBB0_10
```

Example

The image displays a C++ source code editor on the left and its corresponding assembly output on the right, generated by x86-64 clang 15.0.0 with the -O2 -march=skylake flag.

C++ Code (Left):

```
17 const uint32_t NumVtxx {
18     array<uint32_t, N> Weights = {0};
19     array<uint32_t, N> AdjMasks = {0};
20     copy(begin(InW), end(InW), begin(Weights));
21     copy(begin(InM), end(InM), begin(AdjMasks));
22
23     const auto GetSubsetWeight = [&](uint32_t Subset) {
24         uint32_t RV = 0;
25         for (size_t I = 0; I < N; ++I)
26             RV += (Subset & (1 << I)) ? Weights[I] : 0;
27         return RV;
28     };
29
30     const auto IsCompatible = [&](uint32_t Subset) {
31         uint32_t IncompatMask = 0;
32         for (size_t I = 0; I < N; ++I)
33             IncompatMask |= (Subset & (1 << I)) ? AdjMasks[I] : 0;
34         return (Subset & IncompatMask) == 0;
35     };
36
37     uint32_t BestWeight = 0;
38     uint32_t BestSubset = 0;
39     const uint32_t NumSubsets = 1 << NumVtxx;
40     for (uint32_t Subset = 1; Subset < NumSubsets; ++Subset) {
41         const uint32_t Weight = GetSubsetWeight(Subset);
42         if (Weight <= BestWeight)
43             continue;
44
45         if (!IsCompatible(Subset))
46             continue;
47
48         BestWeight = Weight;
49         BestSubset = Subset;
50     }
51     return BestSubset;
52 }
```

Assembly Code (Right):

```
59 vpxor xmm6, xmm6, xmm6
60 xor eax, eax
61 jmp .LBB0_8
62 .LBB0_10: # in Loop: Header=BB0_8 Depth=1
63 inc ecx
64 shrq esi, ecx, ebx
65 test esi, esi
66 jne .LBB0_6
67 .LBB0_8: # =>This Inner Loop Header: Depth=1
68 vmovd xmm7, ecx
69 vpbroadcastq ymm7, xmm7
70 vpand ymm8, ymm7, ymm4
71 vpand ymm7, ymm7, ymm5
72 vpcmpq ymm9, ymm7, ymm5
73 vpand ymm9, ymm9, ymm1
74 vpcmpq ymm10, ymm8, ymm4
75 vpand ymm10, ymm10, ymm0
76 vpaddq ymm9, ymm10, ymm9
77 vextracti128 xmm2, ymm9, 1
78 vpaddq xmm2, xmm9, xmm2
79 vpslufd xmm3, xmm2, 238 # xmm3 = xmm2[2,3,2,3]
80 vpaddq xmm2, xmm2, xmm3
81 vpslufd xmm3, xmm2, 85 # xmm3 = xmm2[1,1,1,1]
82 vpaddq xmm2, xmm2, xmm3
83 vmovd esi, xmm2
84 cmp esi, edx
85 jbe .LBB0_10
86 vpcmpgtd ymm2, ymm8, ymm6
87 vpand ymm2, ymm11, ymm2
88 vpcmpgtd ymm3, ymm7, ymm6
89 vpand ymm3, ymm12, ymm3
90 vpor ymm2, ymm2, ymm3
91 vextracti128 xmm3, ymm2, 1
92 vpor xmm2, xmm2, xmm3
93 vpslufd xmm3, xmm2, 238 # xmm3 = xmm2[2,3,2,3]
94 vpor xmm2, xmm2, xmm3
95 vpslufd xmm3, xmm2, 85 # xmm3 = xmm2[1,1,1,1]
96 vpor xmm2, xmm2, xmm3
97 vmovd edi, xmm2
98 test edi, ecx
99 cmovae eax, ecx
100 cmovae edx, esi
101 jmp .LBB0_10
```

Example

```
C++
17 const uint32_t NumVtx {
18     array<uint32_t, N> Weights = {};
19     array<uint32_t, N> AdjMasks = {};
20     copy(begin(InW), end(InW), begin(Weights));
21     copy(begin(InM), end(InM), begin(AdjMasks));
22
23     const auto GetSubsetWeight = [&](uint32_t Subset) {
24         uint32_t RV = 0;
25         for (size_t I = 0; I < N; ++I)
26             RV += (Subset & (1 << I)) ? Weights[I] : 0;
27         return RV;
28     };
29
30     const auto IsCompatible = [&](uint32_t Subset) {
31         uint32_t IncompatMask = 0;
32         for (size_t I = 0; I < N; ++I)
33             IncompatMask |= (Subset & (1 << I)) ? AdjMasks[I] : 0;
34         return (Subset & IncompatMask) == 0;
35     };
36
37     uint32_t BestWeight = 0;
38     uint32_t BestSubset = 0;
39     const uint32_t NumSubsets = 1 << NumVtx;
40     for (uint32_t Subset = 1; Subset < NumSubsets; ++Subset) {
41         const uint32_t Weight = GetSubsetWeight(Subset);
42         if (Weight <= BestWeight)
43             continue;
44
45         if (!IsCompatible(Subset))
46             continue;
47
48         BestWeight = Weight;
49         BestSubset = Subset;
50     }
51     return BestSubset;
52 }
```

```
x86-64 clang 15.0.0 -O2 -march=skylake
59 vpxor    xmm6, xmm6, xmm6
60 xor      eax, eax
61 jmp      .LBB0_8
62 .LBB0_10:                                # in L
63 inc      ecx
64 shrq     esi, ecx, ebx
65 test     esi, esi
66 jne      .LBB0_6
67 .LBB0_8:                                # =>This
68 vmovd    xmm7, ecx
69 vpbroadcastq ymm7, xmm7
70 vpand     ymm8, ymm7, ymm4
71 vpand     ymm7, ymm7, ymm5
72 vpcmpq    ymm9, ymm7, ymm5
73 vpand     ymm9, ymm9, ymm1
74 vpcmpq    ymm10, ymm8, ymm4
75 vpand     ymm10, ymm10, ymm0
76 vpaddd    ymm9, ymm10, ymm9
77 vextracti128 xmm2, ymm9, 1
78 vpaddd    xmm2, xmm9, xmm2
79 vpslufd   xmm3, xmm2, 238
80 vpaddd    xmm2, xmm2, xmm3
81 vpslufd   xmm3, xmm2, 85
82 vpaddd    xmm2, xmm2, xmm3
83 vmovd     esi, xmm2
84 cmp       esi, edx
85 jbe      .LBB0_10
86 vpcmpgtd  ymm2, ymm8, ymm6
87 vpand     ymm2, ymm11, ymm2
88 vpcmpgtd  ymm3, ymm7, ymm6
89 vpand     ymm3, ymm12, ymm3
90 vpor      ymm2, ymm2, ymm3
91 vextracti128 xmm3, ymm2, 1
92 vpor      xmm2, xmm2, xmm3
93 vpslufd   xmm3, xmm2, 238
94 vpor      xmm2, xmm2, xmm3
95 vpslufd   xmm3, xmm2, 85
96 vpor      xmm2, xmm2, xmm3
97 vmovd     edi, xmm2
98 test      edi, ecx
99 cmov     eax, ecx
100 cmov     edx, esi
101 jmp      .LBB0_10
```

no loops

Example

```
C++
17 const uint32_t NumVtx {
18     array<uint32_t, N> Weights = {0};
19     array<uint32_t, N> AdjMasks = {0};
20     copy(begin(InW), end(InW), begin(Weights));
21     copy(begin(InM), end(InM), begin(AdjMasks));
22
23     const auto GetSubsetWeight = [&](uint32_t Subset) {
24         uint32_t RV = 0;
25         for (size_t I = 0; I < N; ++I)
26             RV += (Subset & (1 << I)) ? Weights[I] : 0;
27         return RV;
28     };
29
30     const auto IsCompatible = [&](uint32_t Subset) {
31         uint32_t IncompatMask = 0;
32         for (size_t I = 0; I < N; ++I)
33             IncompatMask |= (Subset & (1 << I)) ? AdjMasks[I] : 0;
34         return (Subset & IncompatMask) == 0;
35     };
36
37     uint32_t BestWeight = 0;
38     uint32_t BestSubset = 0;
39     const uint32_t NumSubsets = 1 << NumVtx;
40     for (uint32_t Subset = 1; Subset < NumSubsets; ++Subset) {
41         const uint32_t Weight = GetSubsetWeight(Subset);
42         if (Weight <= BestWeight)
43             continue;
44
45         if (!IsCompatible(Subset))
46             continue;
47
48         BestWeight = Weight;
49         BestSubset = Subset;
50     }
51     return BestSubset;
52 }
```

```
x86-64 clang 15.0.0 -O2 -march=skylake
59 vpxor    xmm6, xmm6, xmm6
60 xor     eax, eax
61 jmp     .LBB0_8
62 .LBB0_10:                                # in L
63 inc     ecx
64 shr     esi, ecx, ebx
65 test    esi, esi
66 jne     .LBB0_6
67 .LBB0_8:                                # =>This
68 vmovd    xmm7, ecx
69 vpbroadcastq ymm7, xmm7
70 vpand    ymm8, ymm7, ymm4
71 vpand    ymm7, ymm7, ymm5
72 vpcmpq    ymm9, ymm7, ymm5
73 vpand    ymm9, ymm9, ymm1
74 vpcmpq    ymm10, ymm8, ymm4
75 vpand    ymm10, ymm10, ymm0
76 vpaddd    ymm9, ymm10, ymm9
77 vextracti128 xmm2, ymm9, 1
78 vpaddd    xmm2, xmm9, xmm2
79 vpslufd    xmm3, xmm2, 238
80 vpaddd    xmm2, xmm2, xmm3
81 vpslufd    xmm3, xmm2, 85
82 vpaddd    xmm2, xmm2, xmm3
83 vmovd    esi, xmm2
84 cmp     esi, edx
85 jbe     .LBB0_10
86 vpcmpgtd    ymm2, ymm8, ymm6
87 vpand    ymm2, ymm11, ymm2
88 vpcmpgtd    ymm3, ymm7, ymm6
89 vpand    ymm3, ymm12, ymm3
90 vpor     ymm2, ymm2, ymm3
91 vextracti128 xmm3, ymm2, 1
92 vpor     xmm2, xmm2, xmm3
93 vpslufd    xmm3, xmm2, 238
94 vpor     xmm2, xmm2, xmm3
95 vpslufd    xmm3, xmm2, 85
96 vpor     xmm2, xmm2, xmm3
97 vmovd    edi, xmm2
98 test    edi, ecx
99 cmov     eax, ecx
100 cmov     edx, esi
101 jmp     .LBB0_10
```

no loops
no memory reads

Example

```
C++
17 const uint32_t NumVtxx {
18     array<uint32_t, N> Weights = {0};
19     array<uint32_t, N> AdjMasks = {0};
20     copy(begin(InW), end(InW), begin(Weights));
21     copy(begin(InM), end(InM), begin(AdjMasks));
22
23     const auto GetSubsetWeight = [&](uint32_t Subset) {
24         uint32_t RV = 0;
25         for (size_t I = 0; I < N; ++I)
26             RV += (Subset & (1 << I)) ? Weights[I] : 0;
27         return RV;
28     };
29
30     const auto IsCompatible = [&](uint32_t Subset) {
31         uint32_t IncompatMask = 0;
32         for (size_t I = 0; I < N; ++I)
33             IncompatMask |= (Subset & (1 << I)) ? AdjMasks[I] : 0;
34         return (Subset & IncompatMask) == 0;
35     };
36
37     uint32_t BestWeight = 0;
38     uint32_t BestSubset = 0;
39     const uint32_t NumSubsets = 1 << NumVtxx;
40     for (uint32_t Subset = 1; Subset < NumSubsets; ++Subset) {
41         const uint32_t Weight = GetSubsetWeight(Subset);
42         if (Weight <= BestWeight)
43             continue;
44
45         if (!IsCompatible(Subset))
46             continue;
47
48         BestWeight = Weight;
49         BestSubset = Subset;
50     }
51     return BestSubset;
52 }
```

```
x86-64 clang 15.0.0 -O2 -march=skylake
59 vpxor    xmm6, xmm6, xmm6
60 xor      eax, eax
61 jmp      .LBB0_8
62 .LBB0_10:                                # in L
63 inc      ecx
64 shrq     esi, ecx, ebx
65 test     esi, esi
66 jne      .LBB0_6
67 .LBB0_8:                                # =>This
68 vmovd    xmm7, ecx
69 vpbroadcastq ymm7, xmm7
70 vpand     ymm8, ymm7, ymm4
71 vpand     ymm7, ymm7, ymm5
72 vpcmpq    ymm9, ymm7, ymm5
73 vpand     ymm9, ymm9, ymm1
74 vpcmpq    ymm10, ymm8, ymm4
75 vpand     ymm10, ymm10, ymm0
76 vpaddd    ymm9, ymm10, ymm9
77 vextracti128 xmm2, ymm9, 1
78 vpaddd    xmm2, xmm9, xmm2
79 vpsquid   xmm3, xmm2, 238
80 vpaddd    xmm2, xmm2, xmm3
81 vpsquid   xmm3, xmm2, 85
82 vpaddd    xmm2, xmm2, xmm3
83 vmovd     esi, xmm2
84 cmp       esi, edx
85 jbe      .LBB0_10
86 vpcmpgtd  ymm2, ymm8, ymm6
87 vpand     ymm2, ymm11, ymm2
88 vpcmpgtd  ymm3, ymm7, ymm6
89 vpand     ymm3, ymm12, ymm3
90 vpor      ymm2, ymm2, ymm3
91 vextracti128 xmm3, ymm2, 1
92 vpor      xmm2, xmm2, xmm3
93 vpsquid   xmm3, xmm2, 238
94 vpor      xmm2, xmm2, xmm3
95 vpsquid   xmm3, xmm2, 85
96 vpor      xmm2, xmm2, xmm3
97 vmovd     edi, xmm2
98 test      edi, ecx
99 cmov     eax, ecx
100 cmov     edx, esi
101 jmp      .LBB0_10
```

no loops
no memory reads
vectorization
(xmm/ymm)

Example

N = 14	time
clang O2 <ul style="list-style-type: none">• unroll: off• vectorization: off	
clang O2 <ul style="list-style-type: none">• unroll: on• vectorization: off	
clang O2 <ul style="list-style-type: none">• unroll: on• vectorization: on	

Example

N = 14	time
clang O2 <ul style="list-style-type: none">● unroll: off● vectorization: off	668 mcs
clang O2 <ul style="list-style-type: none">● unroll: on● vectorization: off	
clang O2 <ul style="list-style-type: none">● unroll: on● vectorization: on	

Example

N = 14	time
clang O2 <ul style="list-style-type: none">● unroll: off● vectorization: off	668 mcs
clang O2 <ul style="list-style-type: none">● unroll: on● vectorization: off	93 mcs
clang O2 <ul style="list-style-type: none">● unroll: on● vectorization: on	

x 7

Example

N = 14	time
clang O2 • unroll: off • vectorization: off	668 mcs
clang O2 • unroll: on • vectorization: off	93 mcs
clang O2 • unroll: on • vectorization: on	26 mcs

x 7

x 26

Outcome

- Unroll is generally:
 - useful with large trip count
 - harmful with small trip count
- Trust defaults... but...
 - Clang likely unrolls by default (small trip count case suffers)
 - GCC (O2) likely does not unroll by default. Use PGO / pragma.
- Step and unroll count should be power of 2
- Provide a hint for full unroll for GCC (O2)

Thank you


```

int sum(int *a, int s, int b) {

    int x = 0;

    // #pragma GCC unroll 8

    for (int i = s; i < b; ++i)

        x += a[i];

    return x;

}

```

```

int sum(int *a, int n) {
    int x = 0;
    // #pragma GCC unroll 8
    for (int i = 0; i < n; ++i)
        x += a[i];
    return x;
}

```

```

int sum(int *a, int n) {
    int sums[4] = {0, 0, 0, 0};

    #pragma GCC unroll(1)
    for (int i = 0; i < n; i += 4) {
        #pragma GCC unroll(4)
        for (int k = 0; k < 4; ++k)
            sums[k] += a[k + i];
    }

    int x = 0;
    #pragma GCC unroll(4)
    for (int k = 0; k < 4; ++k)
        x += sums[k];
    return x;
}

```

-O2 -march=skylake -fno-tree-vectorize

-O2 -march=skylake -fno-vectorize -fno-slp-vectorize