Optimizing Your Code Using LLVM

Helping the Compiler Help You

Session 408

Jim Grosbach
Manager, LLVM CPU Team

Roadmap

What’s on tap?

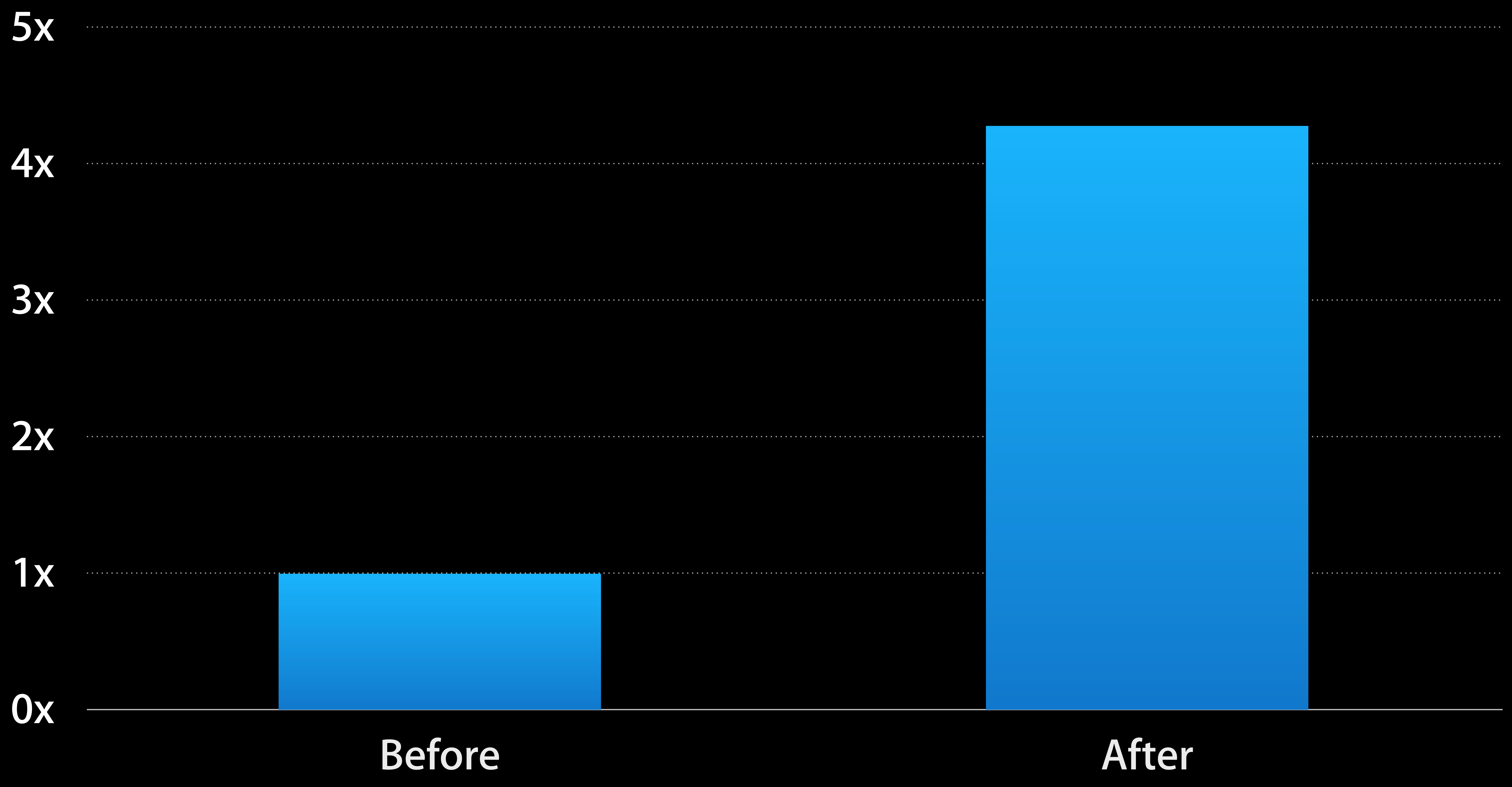
- Xcode and LLVM
- LLVM optimizations and your code
- New and improved LLVM in Xcode 5!

Xcode and the LLVM Compiler

Building and optimizing SciMark-2

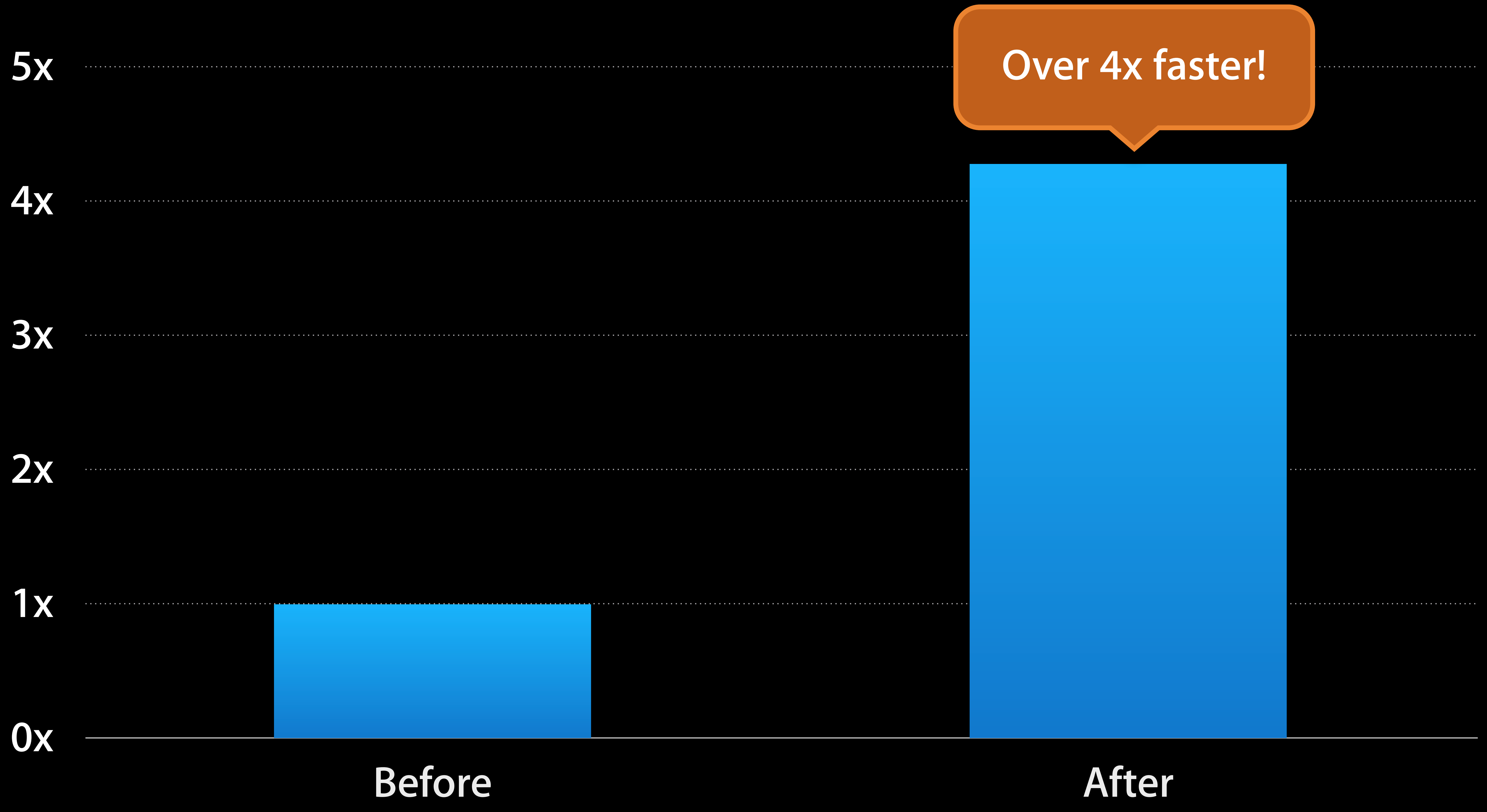
SciMark-2

Measuring optimization impact



SciMark-2

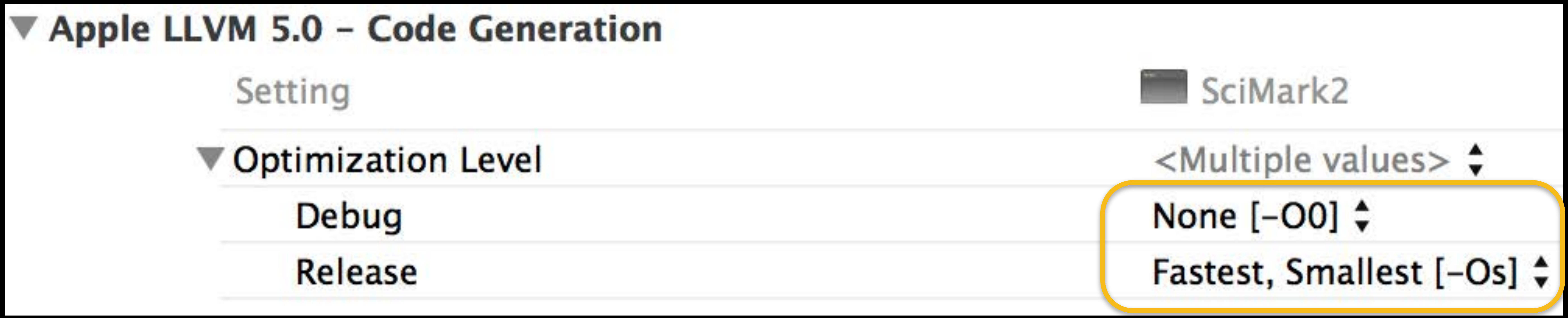
Measuring optimization impact



Xcode Build Settings

Configurations and Optimization

Compiler command line options

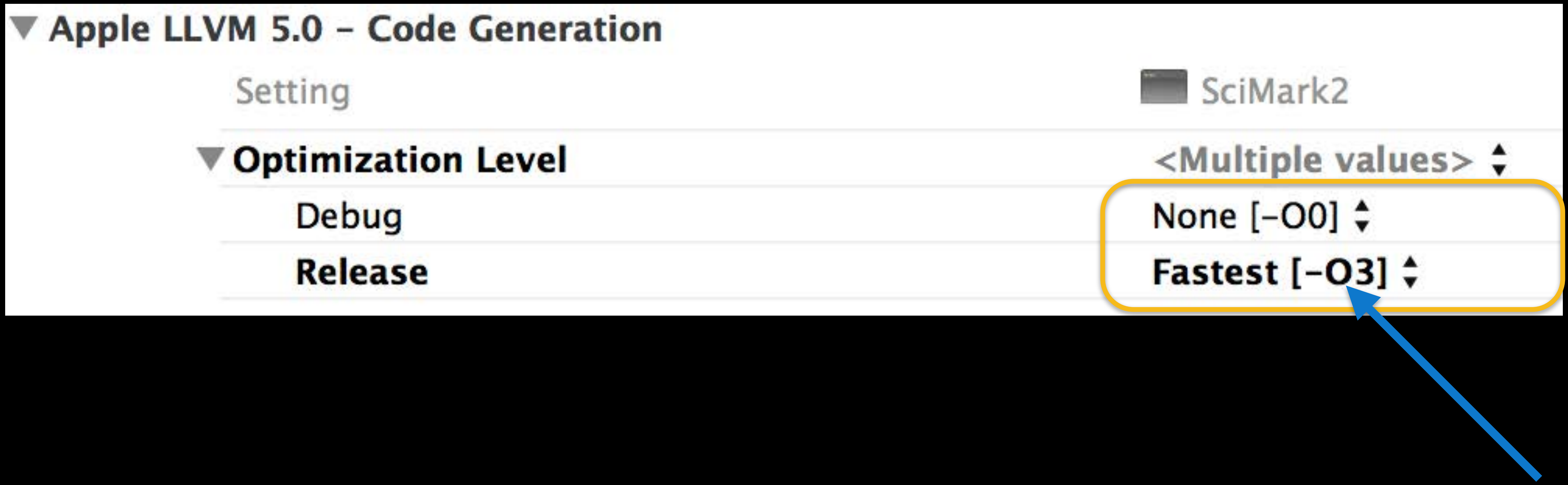


Apple LLVM 5.0 – Code Generation

Setting	SciMark2
Optimization Level	<Multiple values>
Debug	None [-O0]
Release	Fastest, Smallest [-Os]

Build Settings and Optimization

Fine tuning for computationally intensive code

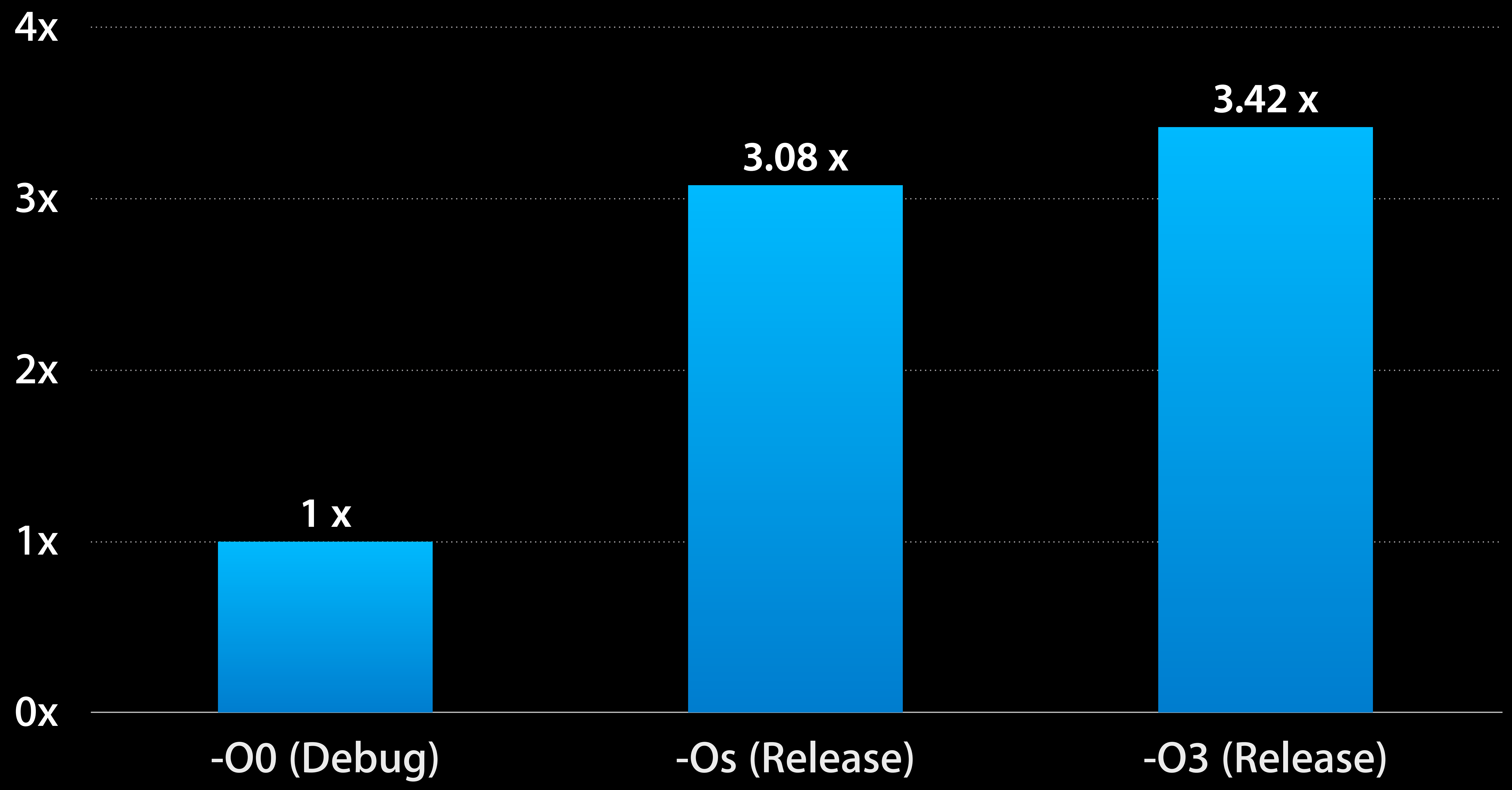


▼ Apple LLVM 5.0 – Code Generation

Setting	SciMark2
▼ Optimization Level	<Multiple values> ⬍
Debug	None [-O0] ⬍
Release	**Fastest [-O3]** ⬍

SciMark-2

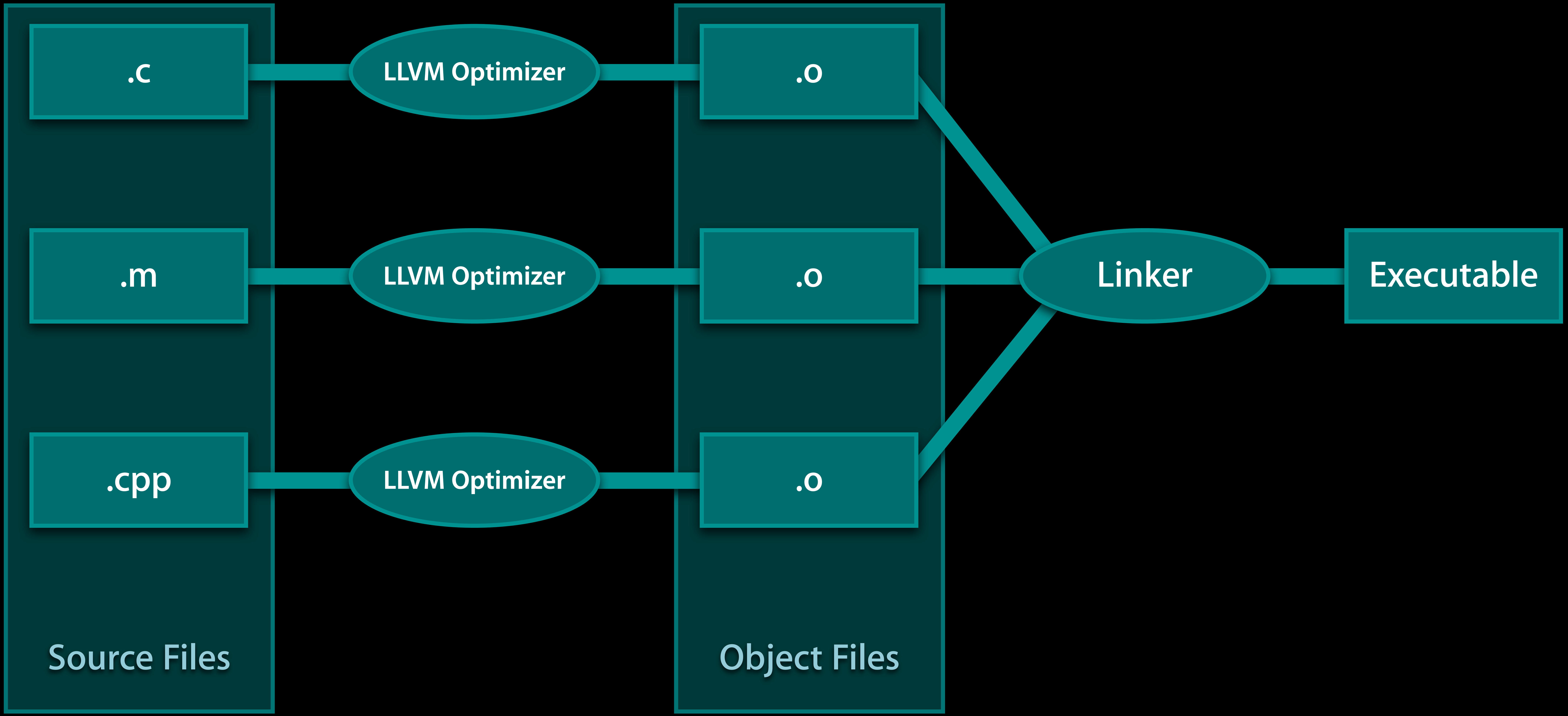
Measuring the effect of -O3



Link-Time Optimization

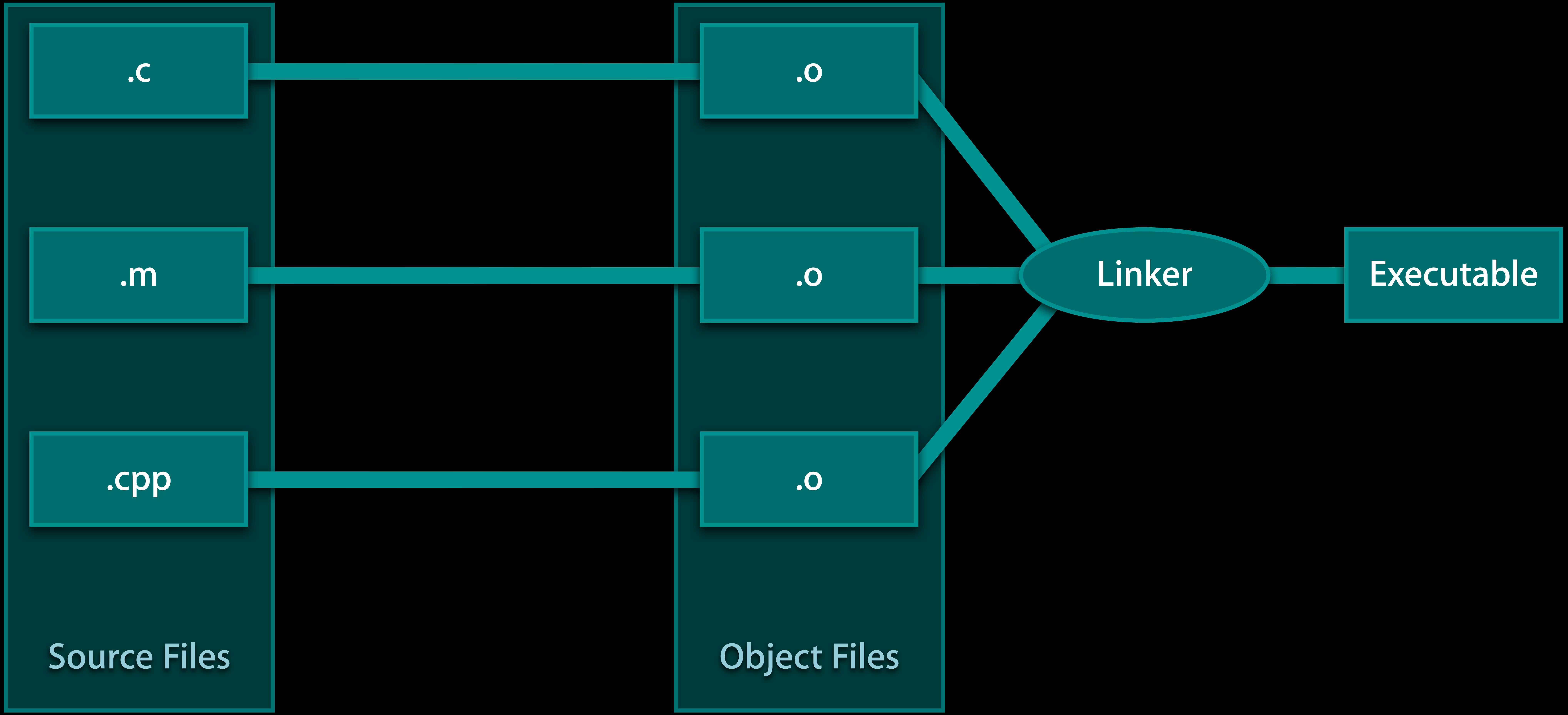
Traditional Optimization

One source file at a time



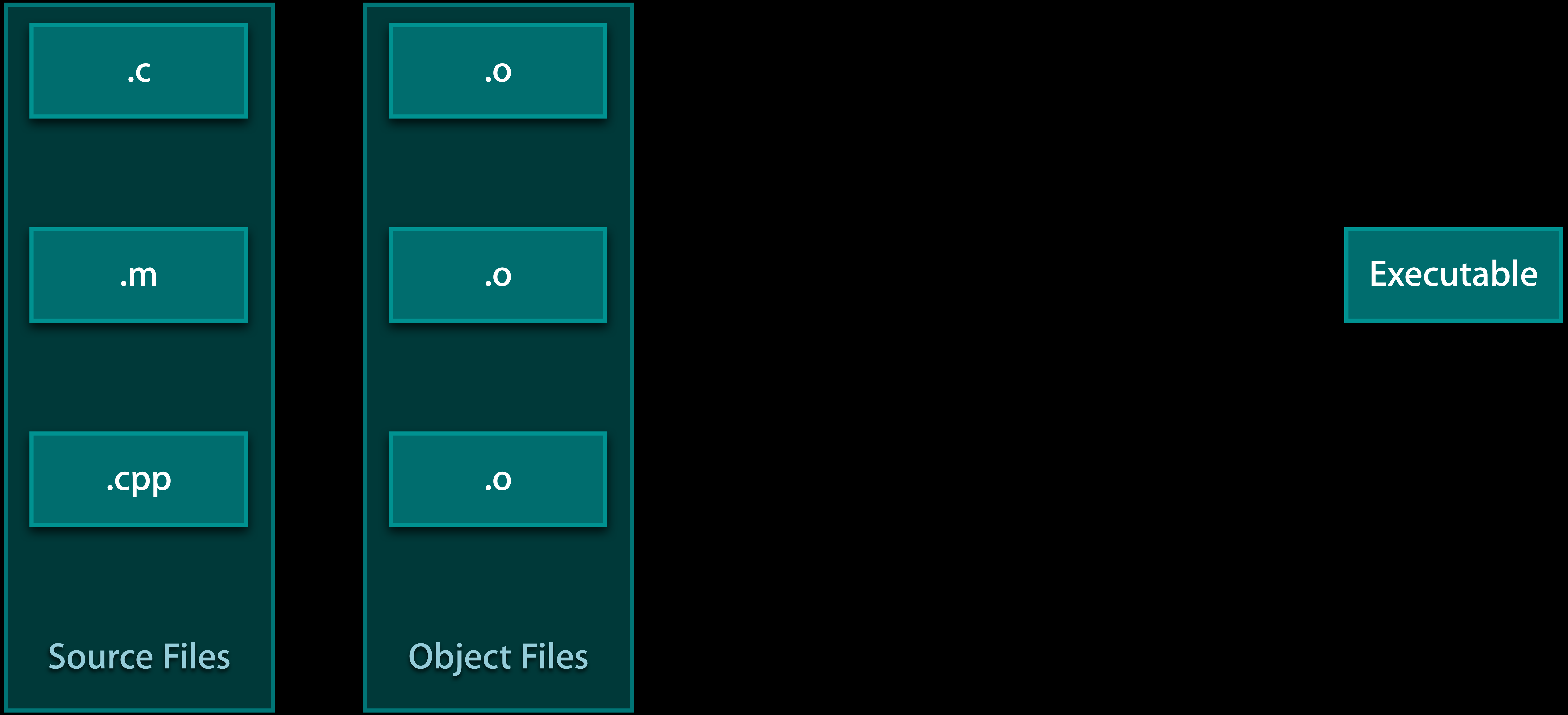
Traditional Optimization

One source file at a time



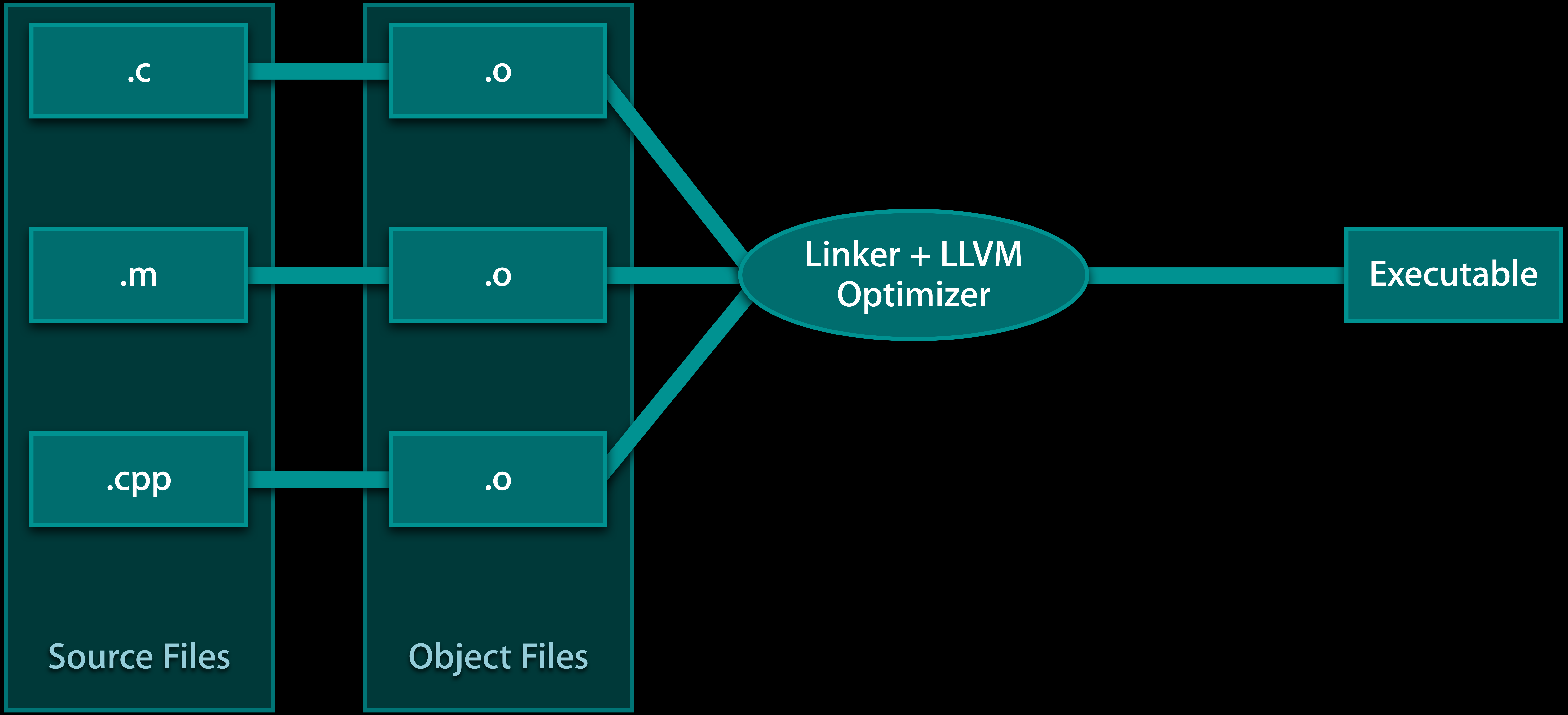
Link-Time Optimization

Whole program at a time



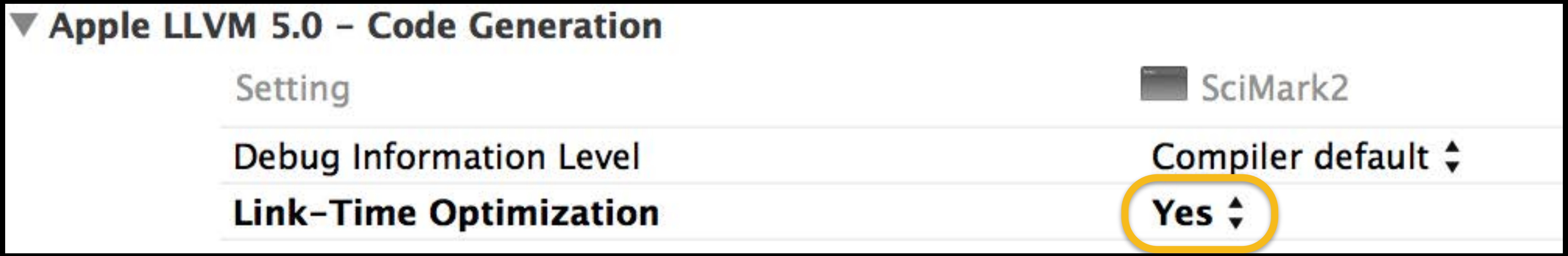
Link-Time Optimization

Whole program at a time



Link-Time Optimization

Compiler option -flto

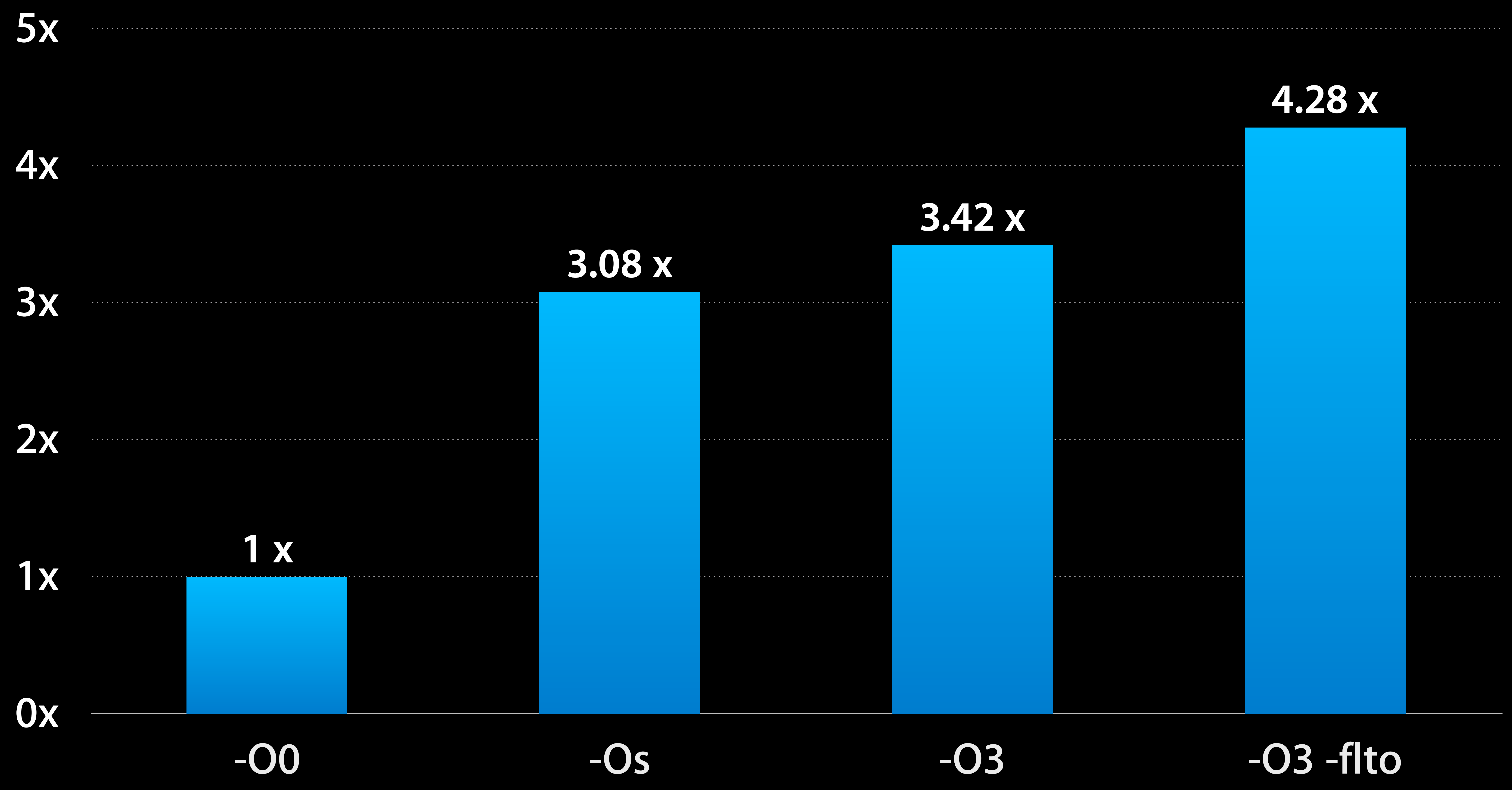


▼ Apple LLVM 5.0 – Code Generation

Setting	SciMark2
Debug Information Level	Compiler default ⬍
Link–Time Optimization	**[✓ Yes ⬍]**

SciMark-2

Measuring the effect of LTO



LLVM Optimizations and Your Code

Pointers

Aliasing

- Can two pointers refer to the same underlying object?
- Optimize memory references
  - Reorder loads and stores
  - Remove redundant loads
  - Remove dead stores

Strict Aliasing

-fstrict-aliasing

```
void foo(float *a, float *b, int *c) {
  *a = *c;
  *b = *c;
}
```

Strict Aliasing

-fstrict-aliasing

```
void foo(float *a, float *b, int *c) {
  *a = *c;
  *b = *c;
}
```

```
_foo:
  vldr  s0, [r2]          // Load ‘c’ only once
  vcvt.f32.s32 d0, d0    // Convert to float
  vstr s0, [r0]           // Store result to ‘a’
  vstr s0, [r1]           // Store result to ‘b’
  bx    lr
```

Aliasing Pointers

Type information isn’t always enough

```
void foo(int *a, int *b, int *c) {
  *a = *c;
  *b = *c;
}
```

Aliasing Pointers

Type information isn’t always enough

```
void foo(int *a, int *b, int *c) {
  *a = *c;
  *b = *c;
}
```

```
_foo:
  ldr r3, [r2]  // load ‘c’
  str r3, [r0]  // store to ‘a’
  ldr r0, [r2]  // load ‘c’ again
  str r0, [r1]  // store to ‘b’
  bx  lr
```

Restricted Pointers

- Specified via the `restrict` keyword
- Object can’t be aliased in its scope

‘restrict’

```
void foo(int *a, int *b, int *c) {
  *a = *c;
  *b = *c;
}
```

```
_foo:
  ldr r2, [r2]  // load ‘c’
  str r2, [r0]  // store to ‘a’
  ldr r2, [r2]  // load ‘c’ again
  str r2, [r1]  // store to ‘b’
  bx  lr
```

‘restrict’

```
void foo(int *a, int *b, int *restrict c) {
  *a = *c;
  *b = *c;
}
```

```
_foo:
  ldr r2, [r2]  // load ‘c’
  str r2, [r0]  // store to ‘a’
  str r2, [r1]  // store to ‘b’
  bx  lr
```

Floating Point Optimization

Floating Point Math

- Representation challenges
  - -0.0 vs. 0.0
  - NaN (Not a Number)
- Reassociation

`(a + b) + c` may not equal `a + (b + c)`

Floating Point Associativity

- Re-association is not completely safe for floating-point

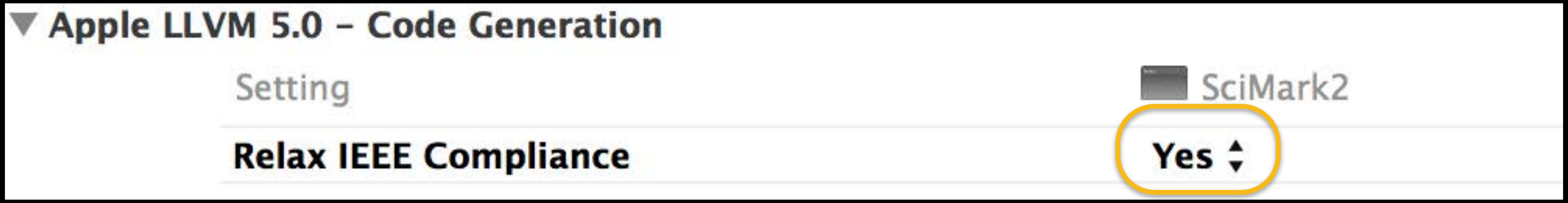
```
a = 1234.567
b = 45.67834
c = 0.0004

(a + b) + c = 1280.245
a + (b + c) = 1280.246
```

Fast Math

-ffast-math

- Enable aggressive floating point optimization



▼ Apple LLVM 5.0 – Code Generation

Setting	SciMark2
Relax IEEE Compliance	**Yes**

Floating Point Dot Product

```
float vec1[] = ...
float vec2[3] = { 1.25, 0.0, 1.5 };
float prod = 0.0;
prod += vec1[0] * vec2[0];
prod += vec1[1] * vec2[1];
prod += vec1[2] * vec2[2];
```

```
$ xcrun clang -arch armv7s -O3  file.c
```

```
vmul.f32  d22, d2, d16
vmul.f32  d16, d1, d18
vadd.f32  d22, d22, d18
vmul.f32  d18, d0, d20
vadd.f32  d16, d22, d16
vadd.f32  d1, d16, d18
```

Floating Point Dot Product

```
float vec1[] = ...
float vec2[3] = { 1.25, 0.0, 1.5 };
float prod = 0.0;
prod += vec1[0] * vec2[0];
prod += vec1[1] * vec2[1];
prod += vec1[2] * vec2[2];
```

```
$ xcrun clang -arch armv7s -O3 -ffast-math file.c
```

```
vmul.f32  d22, d2, d16
vmul.f32  d16, d1, d18
vadd.f32  d22, d22, d18
vmul.f32  d18, d0, d20
vadd.f32  d16, d22, d16
vadd.f32  d1, d16, d18
```

Floating Point Dot Product

```
float vec1[] = ...
float vec2[3] = { 1.25, 0.0, 1.5 };
float prod = 0.0;
prod += vec1[0] * vec2[0];
prod += vec1[1] * vec2[1];
prod += vec1[2] * vec2[2];
```

```
$ xcrun clang -arch armv7s -O3 -ffast-math file.c
```

```
vmul.f32  d18, d0, d18
vmul.f32  d16, d1, d16
vadd.f32  d1, d18, d16
```

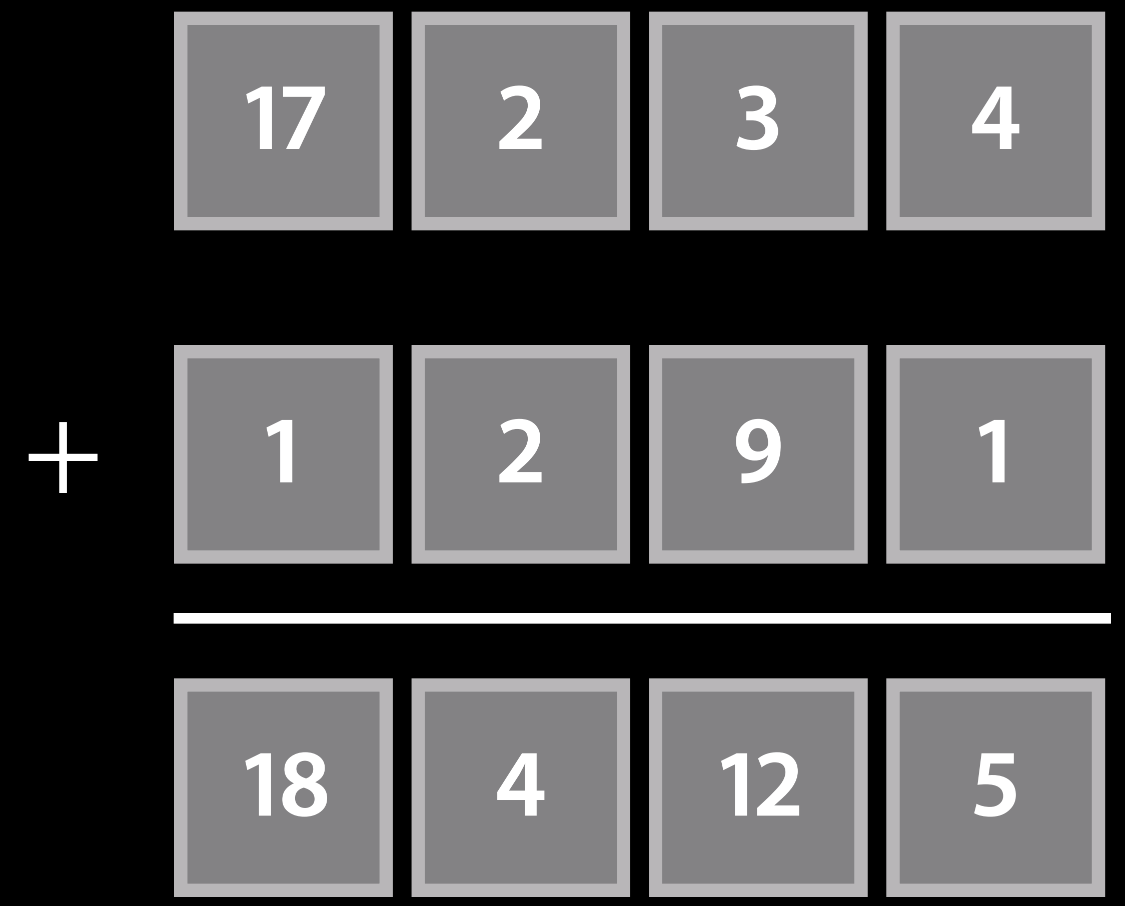
Vectorization

Vector Instruction Sets

- Modern architectures have vector units
  - OS X
    - SSE, SSE2, SSE3, SSE4
    - AVX, AVX2
  - iOS
    - NEON
- We want to use them effectively

Acceleration Using Vectors

- Single Instruction Multiple Data (SIMD)
- Vector instructions operate on multiple values
- Multiple data = fewer instructions to execute

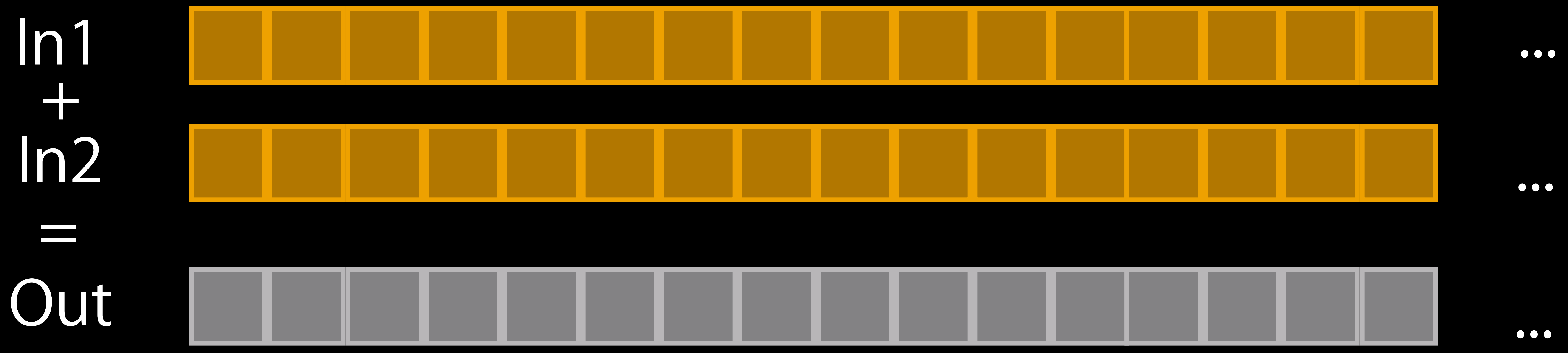


17 2 3 4

+ 1 2 9 1

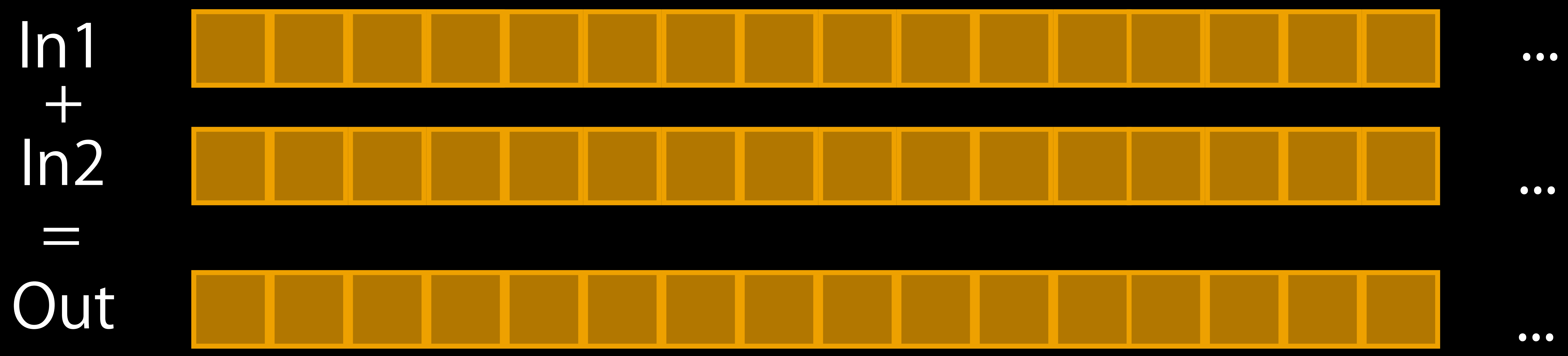
18 4 12 5

Using Vectors for Loops



```
for (int i = 0; i < N; i++) {
  out[i] = in1[i] + in2[i];
}
```

Using Vectors for Loops



In1
+
In2
=
Out

...

...

...

```
for (int i = 0; i < N; i++) {
  out[i] = in1[i] + in2[i];
}
```

Vector Intrinsics

- Intrinsics are functions that represent CPU instructions

```
#include <arm_neon.h>
void do_mul_add(int *a, int *b, int *c, int *out) {
  vst1q_s32(out,
            vaddq_s32(vmulq_s32(vld1q_s32(a),
                                vld1q_s32(b)),
                      vld1q_s32(c)));
}
```

Vector Intrinsics

- Intrinsics are functions that represent CPU instructions

```
#include <arm_neon.h>
void do_mul_add(int *a, int *b, int *c, int *out) {
  vst1q_s32(out,
            vaddq_s32(vmulq_s32(vld1q_s32(a),
                                vld1q_s32(b)),
                      vld1q_s32(c)));
}
```

```
_do_mul_add:
   vld1.32   {d16, d17}, [r1]  // load four integers from ‘b’
   vld1.32   {d18, d19}, [r0]  // load four integers from ‘a’
   vld1.32   {d20, d21}, [r2]  // load four integers from ‘c’
   vmla.i32  q10, q9, q8       // multiply-accumulate
   vst1.32   {d20, d21}, [r3]  // store four integers to ‘out’
   bx  lr
```

Vector Attributes

- LLVM supports target independent vector attributes

```
typedef __attribute__((ext_vector_type(4))) int int4;

void do_mul_add(int4 *a, int4 *b, int4 *c, int4 *out) {
  *out = (*a * *b) + *c;
}
```

Vector Attributes

- LLVM supports target independent vector attributes

```
typedef __attribute__((ext_vector_type(4))) int int4;

void do_mul_add(int4 *a, int4 *b, int4 *c, int4 *out) {
  *out = (*a * *b) + *c;
}
```

```
_do_mul_add:
   vld1.32   {d16, d17}, [r1]  // load four integers from ‘b’
   vld1.32   {d18, d19}, [r0]  // load four integers from ‘a’
   vld1.32   {d20, d21}, [r2]  // load four integers from ‘c’
   vmla.i32  q10, q9, q8       // multiply-accumulate
   vst1.32   {d20, d21}, [r3]  // store four integers to ‘out’
   bx  lr
```

Challenges

- Target dependencies remain
  - Which vector operations are supported
  - Native vector width
- The compiler can do more

Auto-Vectorization in LLVM

Nadav Rotem
Manager, LLVM Performance Team

Auto-Vectorization in LLVM

NEW

- Xcode 5 has a new auto-vectorizer
- Optimizes numeric loops
- Works on all iOS and OS X hardware

Who May Benefit ?

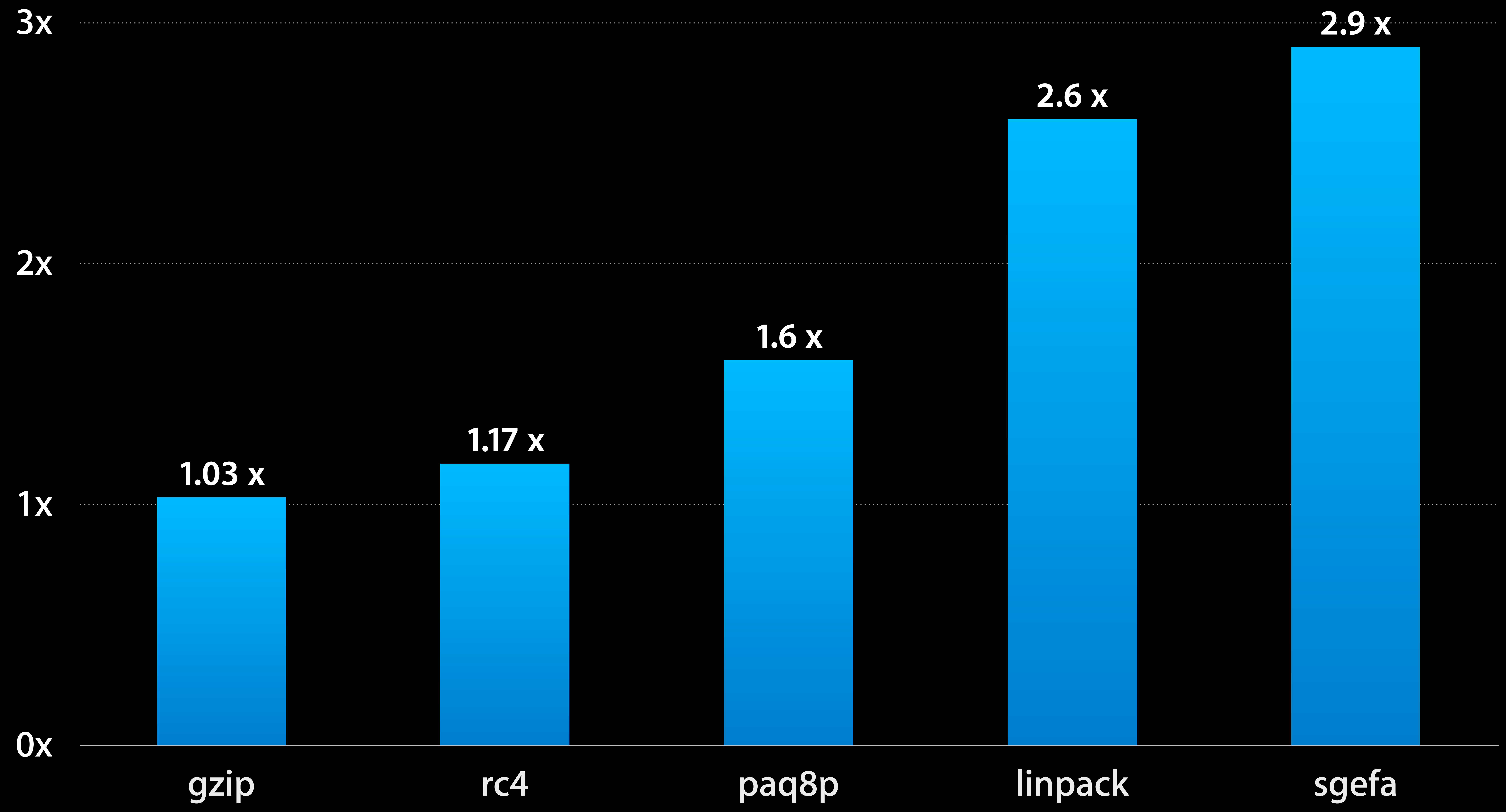
- Small numeric loops:
  - Linear algebra
  - Physics simulations
  - Image processing

✓

- Other types of code:
  - User Interface
  - Parsing text
  - Database queries

✗

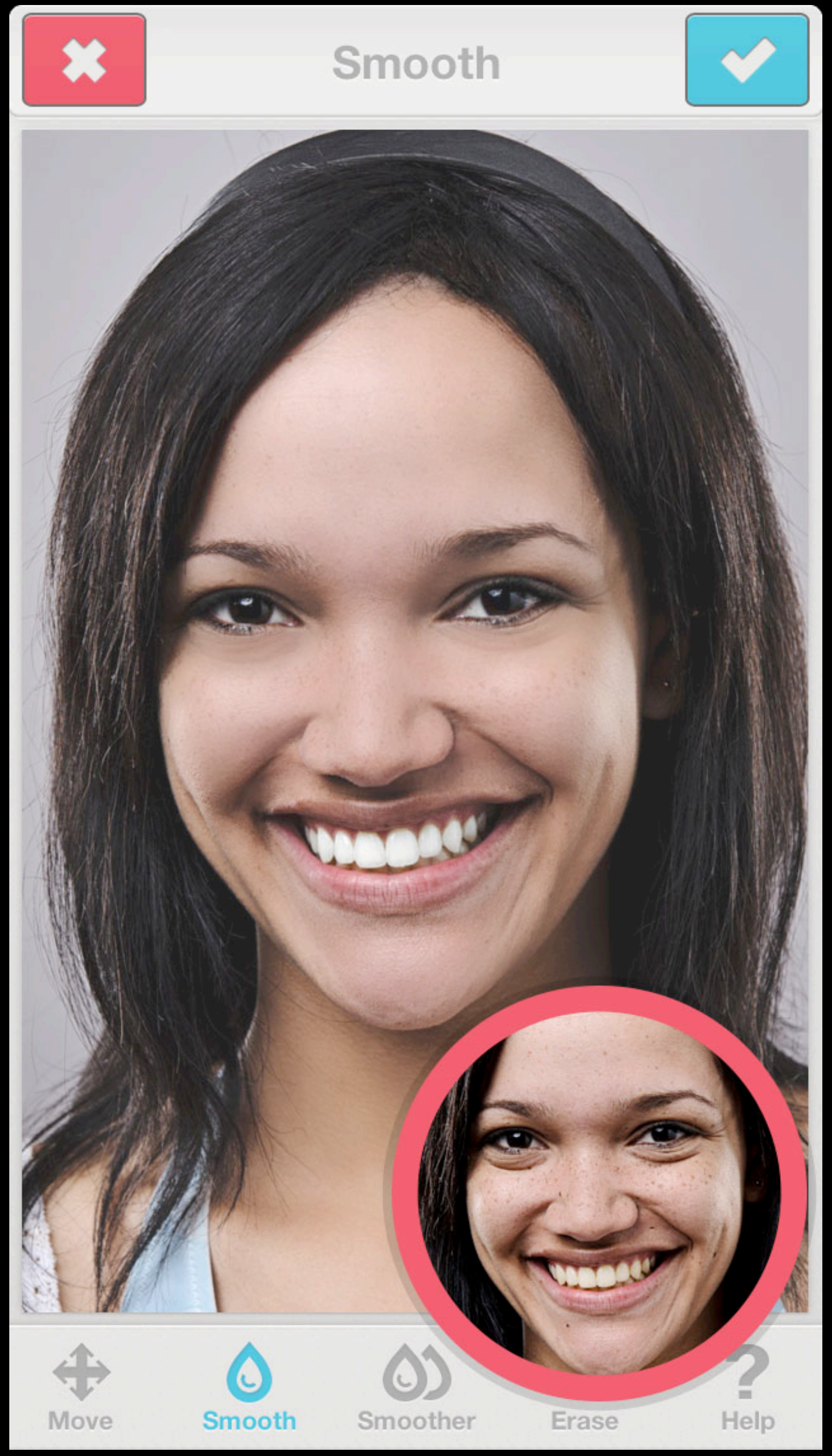
Performance Numbers



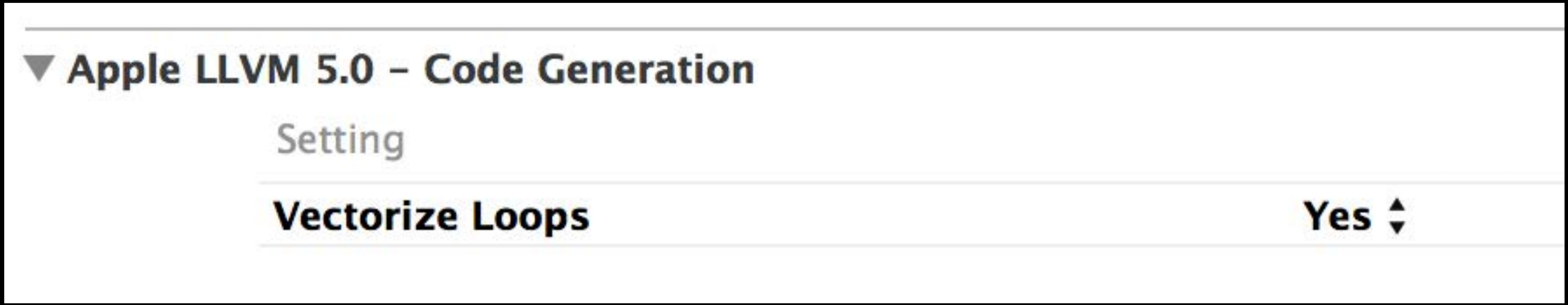
Accelerating Apps

“We've used the vectorizer tool [auto-vectorizer in Apple LLVM 5] to recompile Facetune. Using this new feature, we will be able to reduce our image filtering run time by some 30%! This will obviously improve the experience for our users, making the app smoother and allowing us to run more complex image processing algorithms in the spare time gained.”

facetune



Enabling the Auto-Vectorizer



▼ Apple LLVM 5.0 – Code Generation

Setting	
Vectorize Loops	Yes

```
$ xcrun clang  -O3 -fvectorize file.c
```

Vectorizing Your Code

Example of Generated Code

```
for (int i = 0; i < N; i++) {
  out[i] = in1[i] + in2[i];
}
```

iOS

```
L0:
  vld1.32   {d16, d17}, [r4]
  vld1.32   {d18, d19}, [r5]
  subs.w    r12, r12, #4
  add.w     r4, r4, #16
  add.w     r5, r5, #16
  vadd.f32  q8, q9, q8
  vst1.32   {d16, d17}, [lr]
  add.w     lr, lr, #16
  bne       L0
```

Example of Generated Code

```
for (int i = 0; i < N; i++) {
  out[i] = in1[i] + in2[i];
}
```

OS X

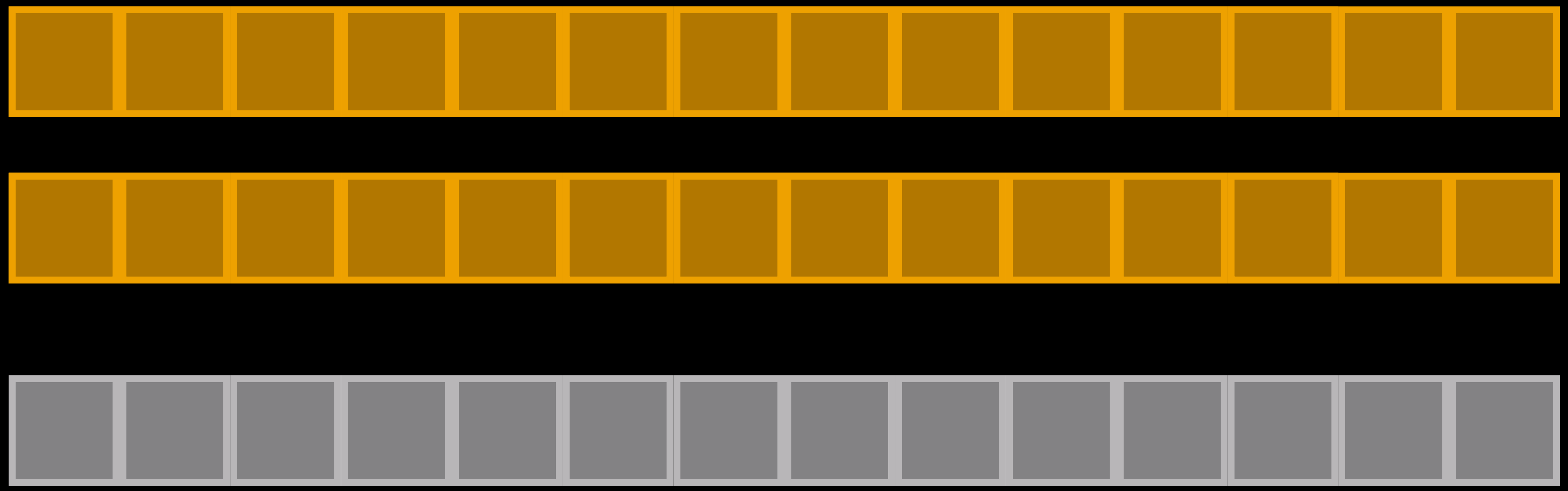
```
L0:
  vmovups (%rdx,%rax,4), %ymm0
  vaddps  (%rsi,%rax,4), %ymm0, %ymm0
  vmovups %ymm0, (%rcx,%rax,4)
  addq    $8, %rax
  cmpq    %rax, %r9
  jne     L0
```

Vectorization is Not Trivial

- The loop below appears to be easily vectorizable
- Need to overcome two problems

```
for (int i = 0; i < N; i++) {
  out[i] = in1[i] + in2[i];
}
```

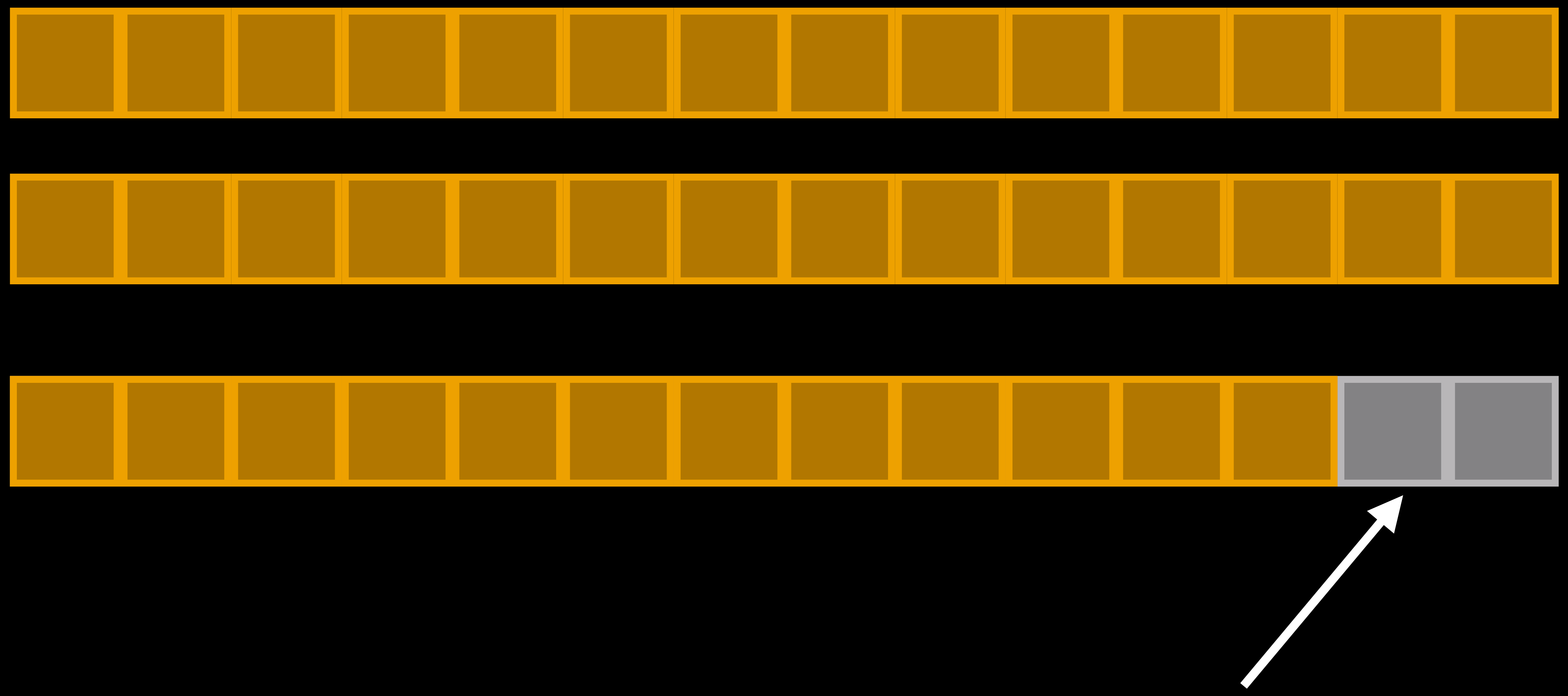
Problem #1 - Remainder Iterations



```
N = 14;

for (int i = 0; i < N; i++) {
  out[i] = in1[i] + in2[i];
}
```

Problem #1 - Remainder Iterations



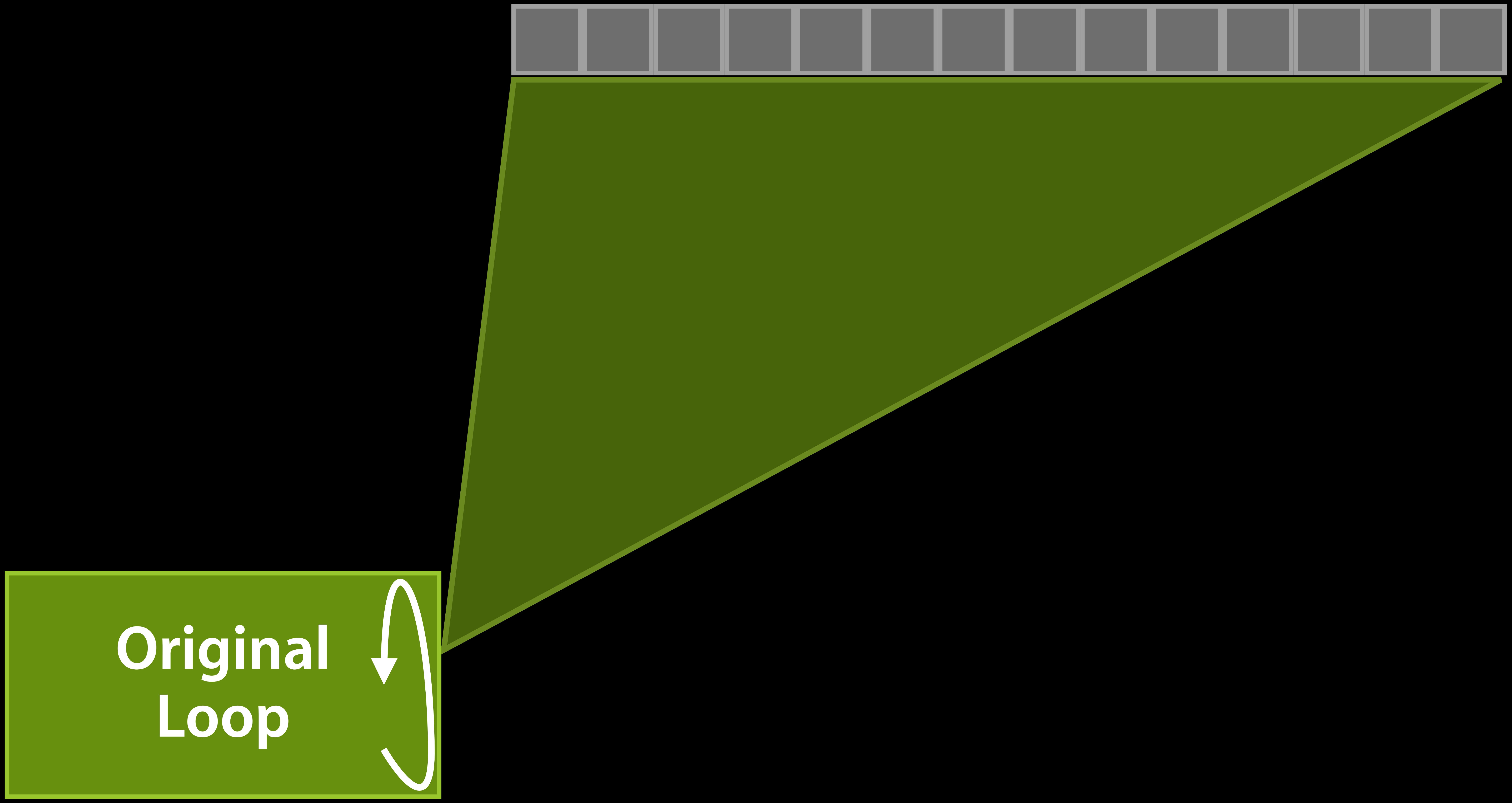
```
N = 14;

for (int i = 0; i < N; i++) {
  out[i] = in1[i] + in2[i];
}
```

Remainder Iterations

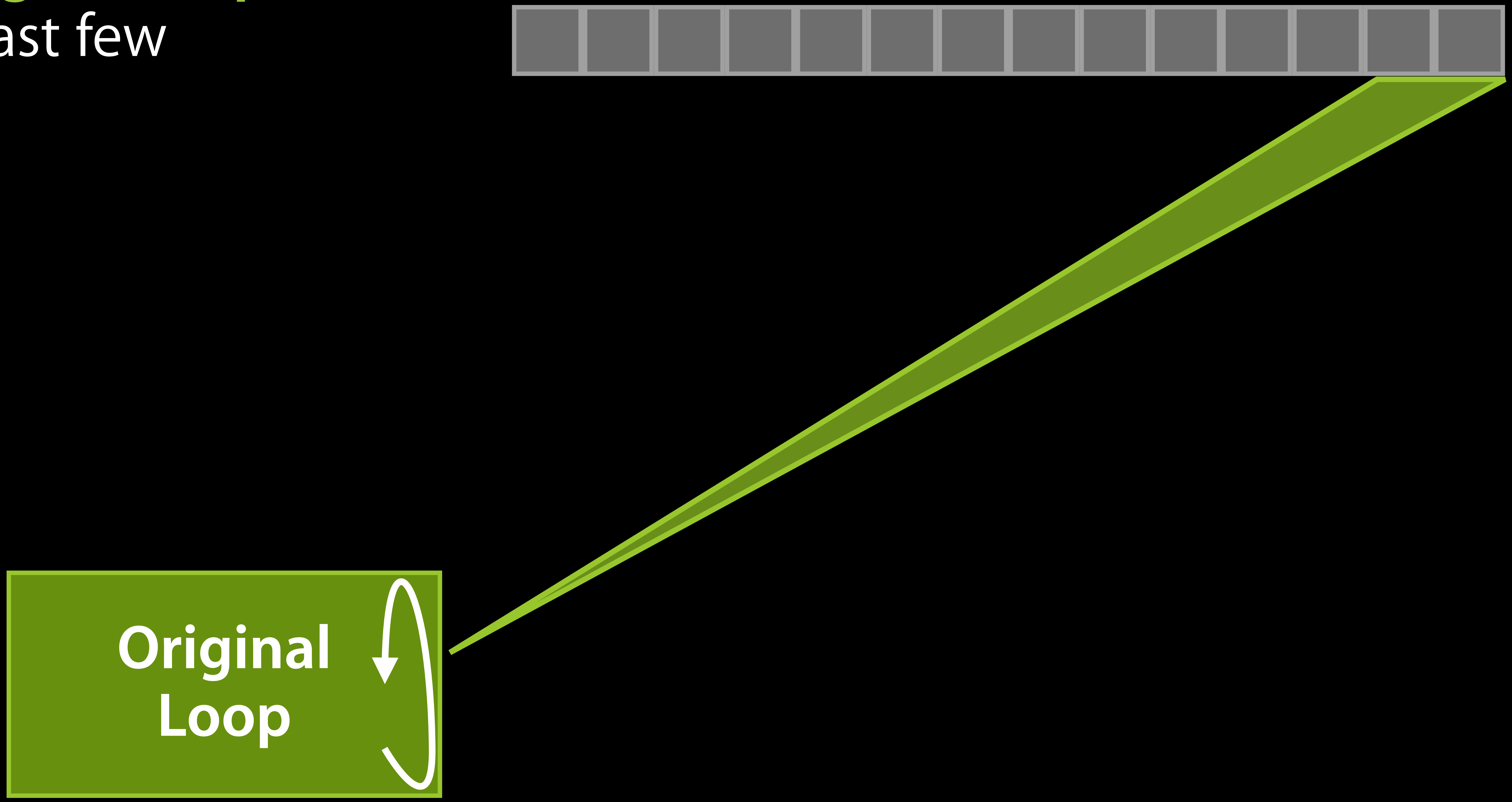
- Number of iterations may not be divisible by vector width
  - `Count = 17`
- Number of iterations does not have to be constant
  - `Count = m`

Handling the Last Few Iterations



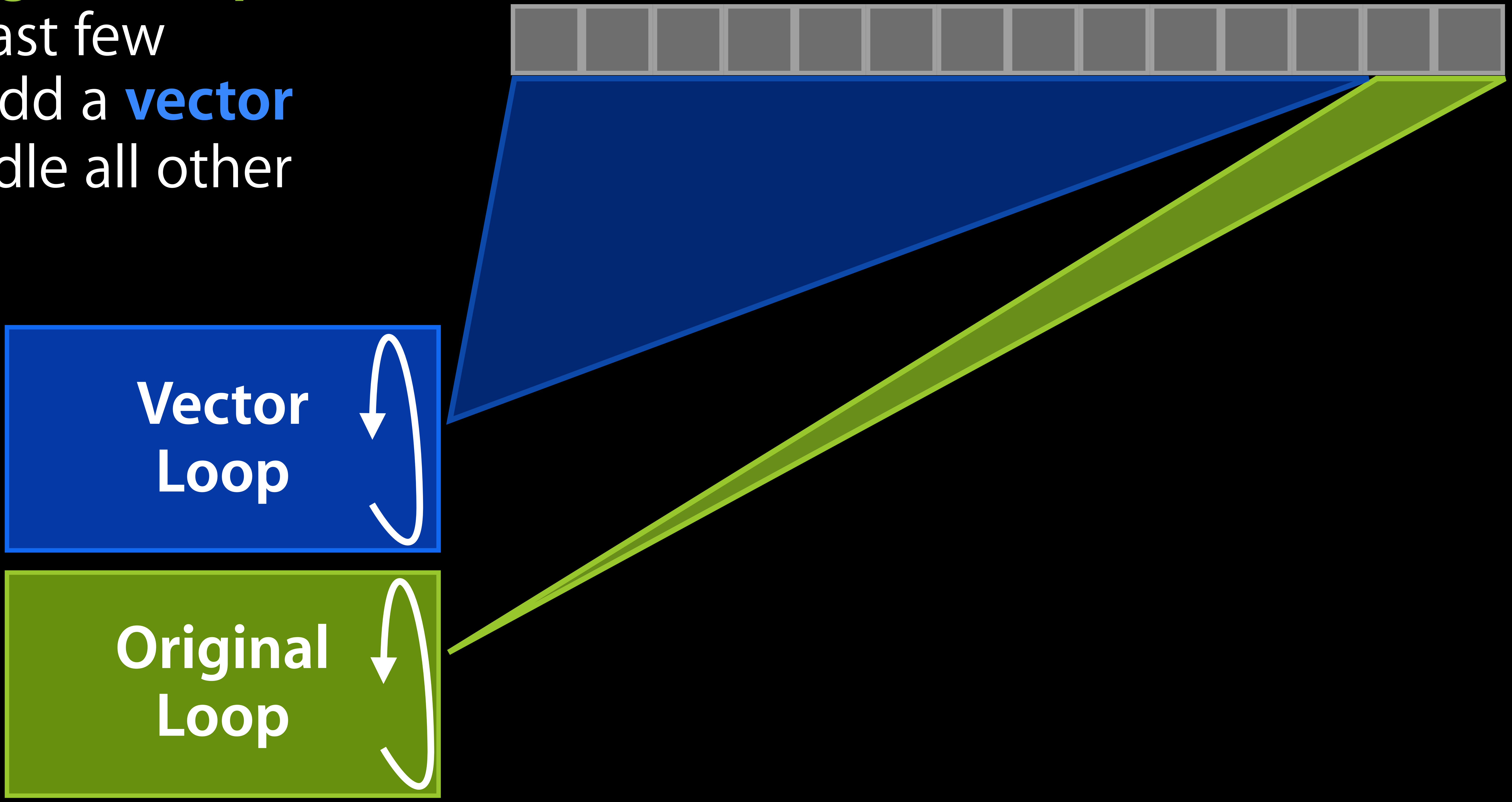
Handling the Last Few Iterations

Keep the **original loop** to handle the last few iterations.



Handling the Last Few Iterations

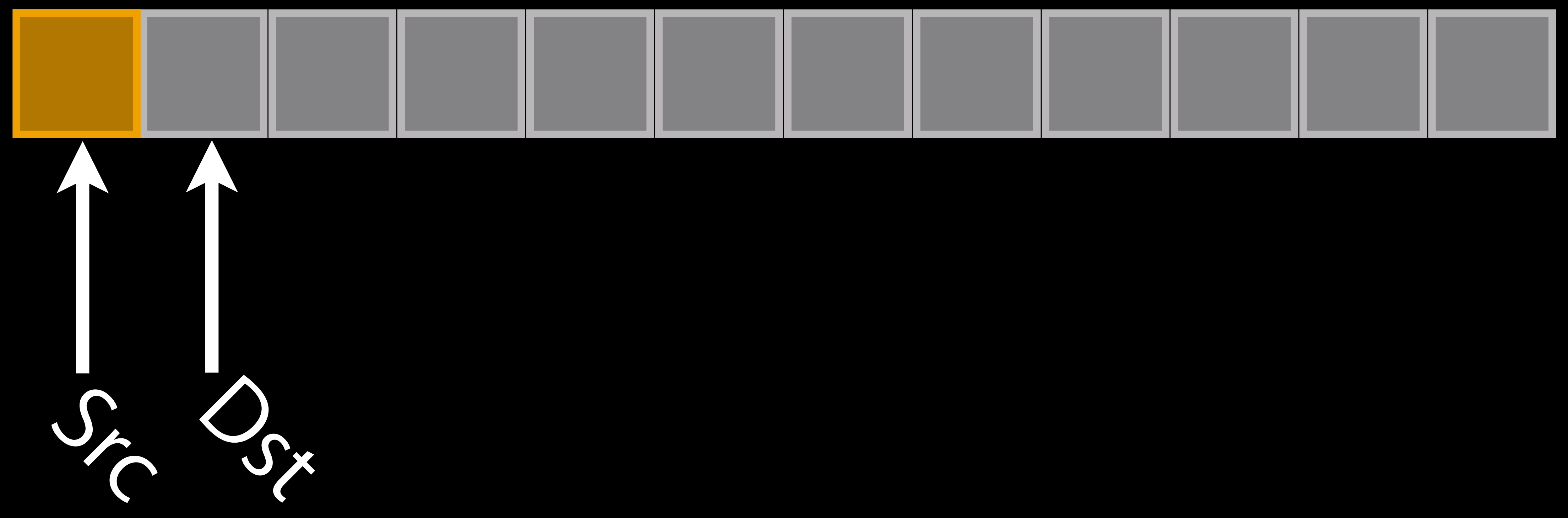
Keep the **original loop** to handle the last few iterations. Add a **vector loop** to handle all other iterations.



Problem #2 - Pointer Safety

- Pointers may point to overlapping memory
- Unsafe to re-order reads and writes

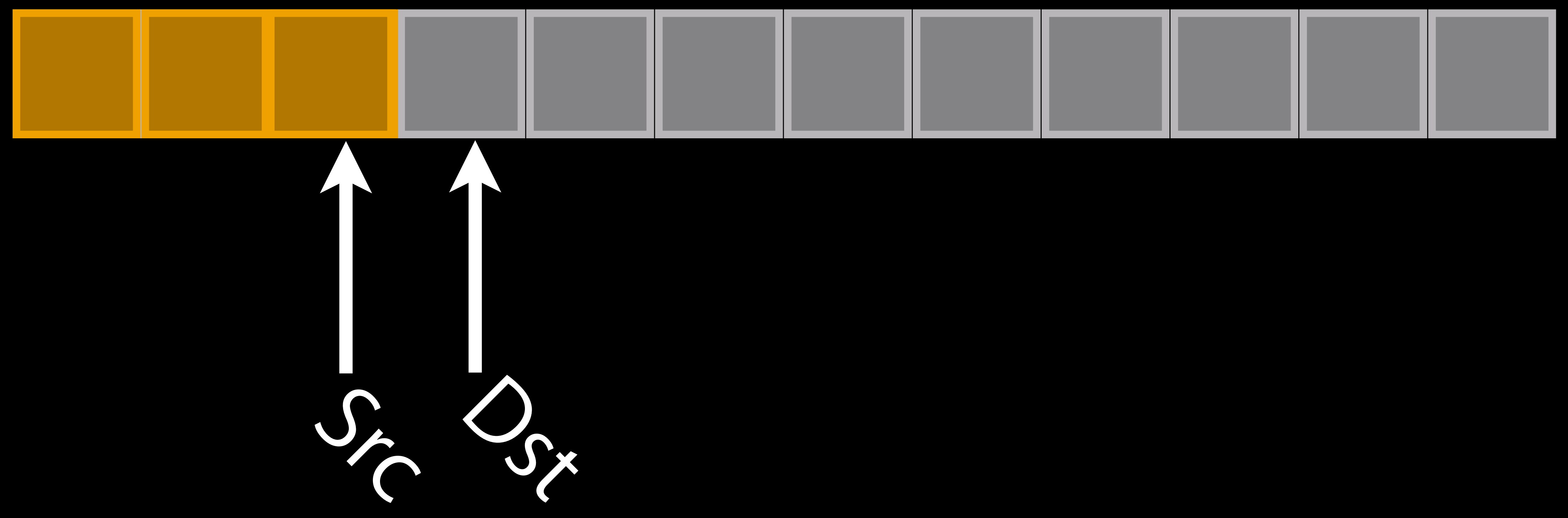
```
for (i = 0; i < n; i++) {
 Dst[i] = Src[i];
}
```



Problem #2 - Pointer Safety

- Pointers may point to overlapping memory
- Unsafe to re-order reads and writes

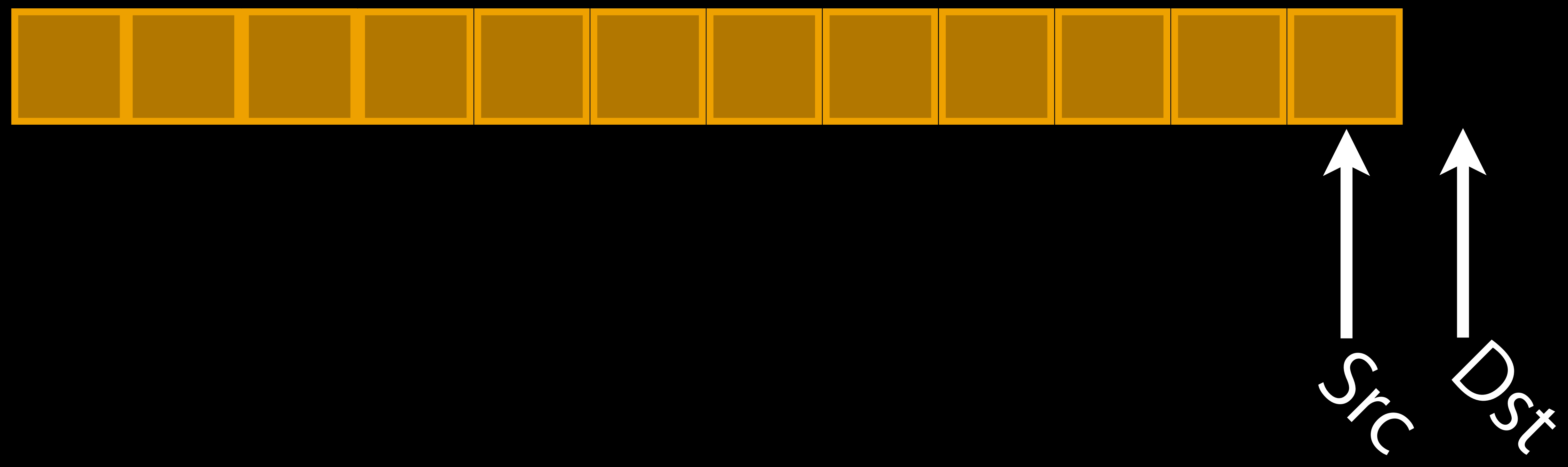
```
for (i = 0; i < n; i++) {
 Dst[i] = Src[i];
}
```



Problem #2 - Pointer Safety

- Pointers may point to overlapping memory
- Unsafe to re-order reads and writes

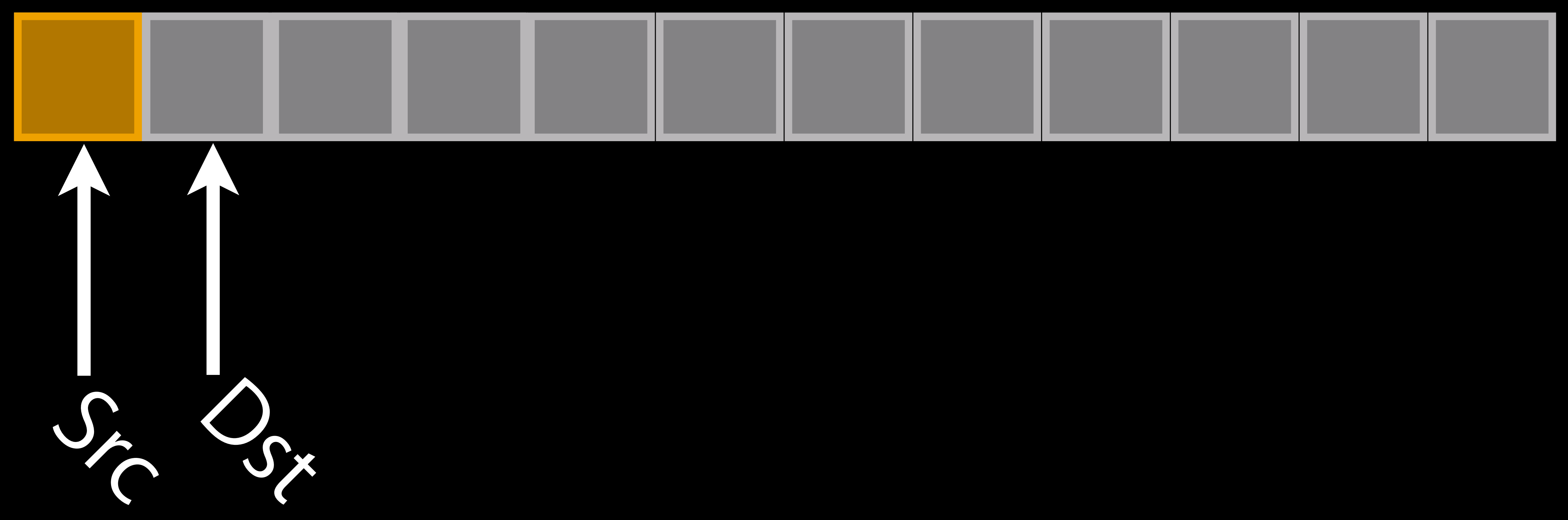
```
for (i = 0; i < n; i++) {
 Dst[i] = Src[i];
}
```



Problem #2 - Pointer Safety

- Pointers may point to overlapping memory
- Unsafe to re-order reads and writes

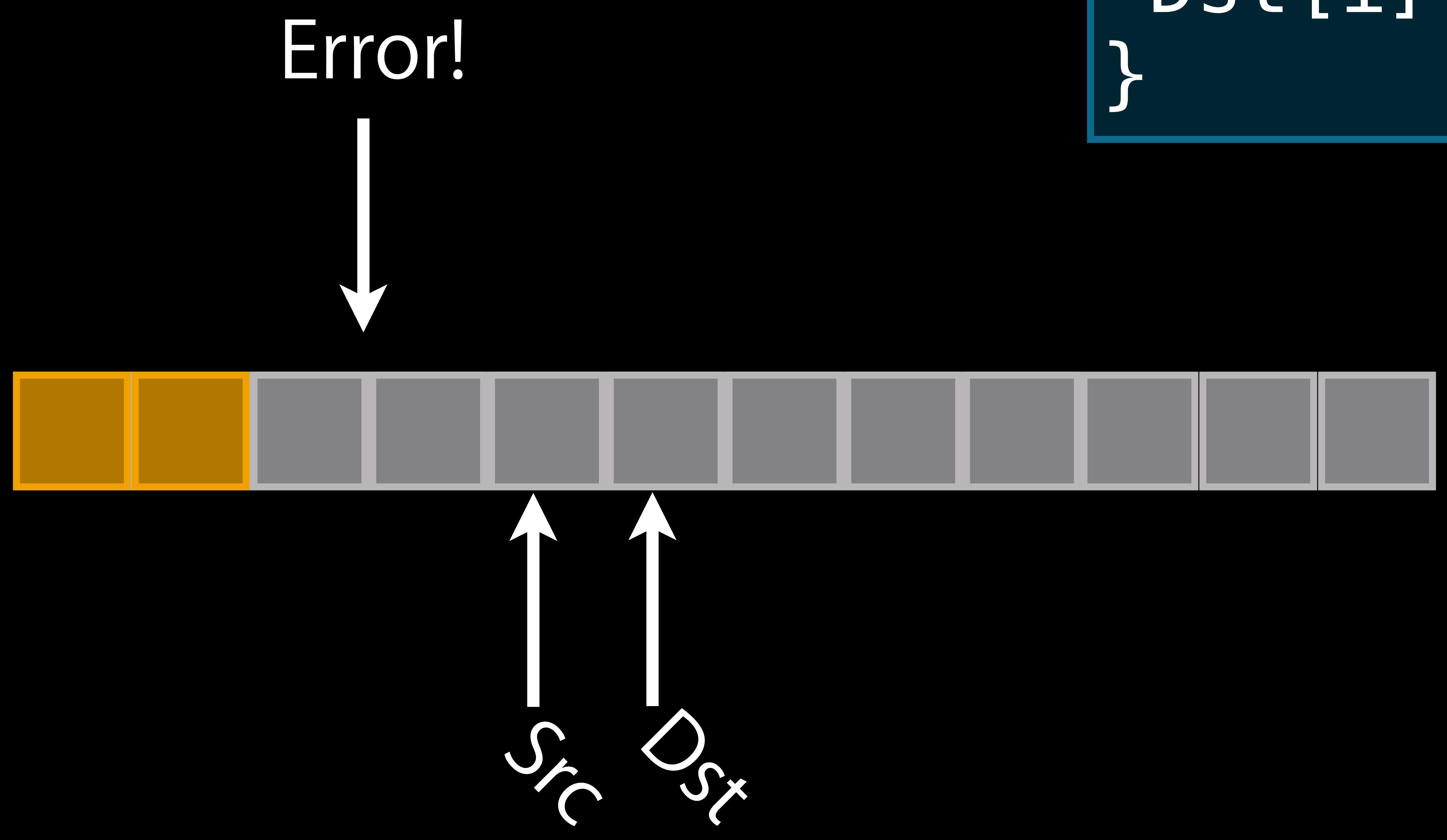
```
for (i = 0; i < n; i++) {
 Dst[i] = Src[i];
}
```



Problem #2 - Pointer Safety

- Pointers may point to overlapping memory
- Unsafe to re-order reads and writes

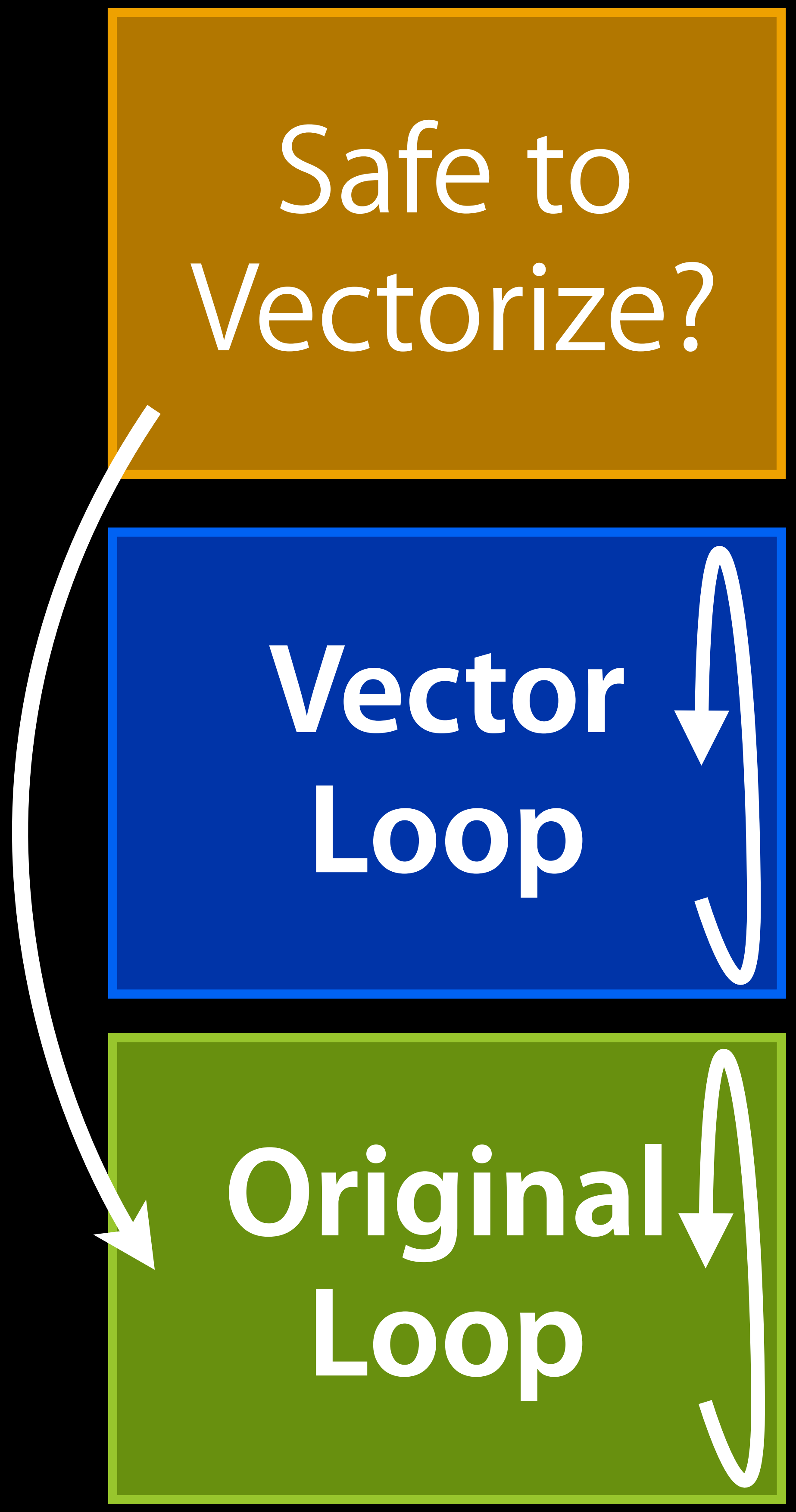
```
for (i = 0; i < n; i++) {
 Dst[i] = Src[i];
}
```



Safety Checks

- Pointer destination often known only at runtime
- Add code to check if safe to vectorize
- Use original loop if unsafe

```
void addFP(int N, float *in1,
                  float *in2,
                  float *out) {
  if (!overlap(in1, out) && !overlap(in2, out))
    for (i = 0; i < N-3; i+=4)  {  ...  }
  for ( ; i < N; i++)  {  ...  }
}
```



Safe to Vectorize?

Vector Loop

Original Loop

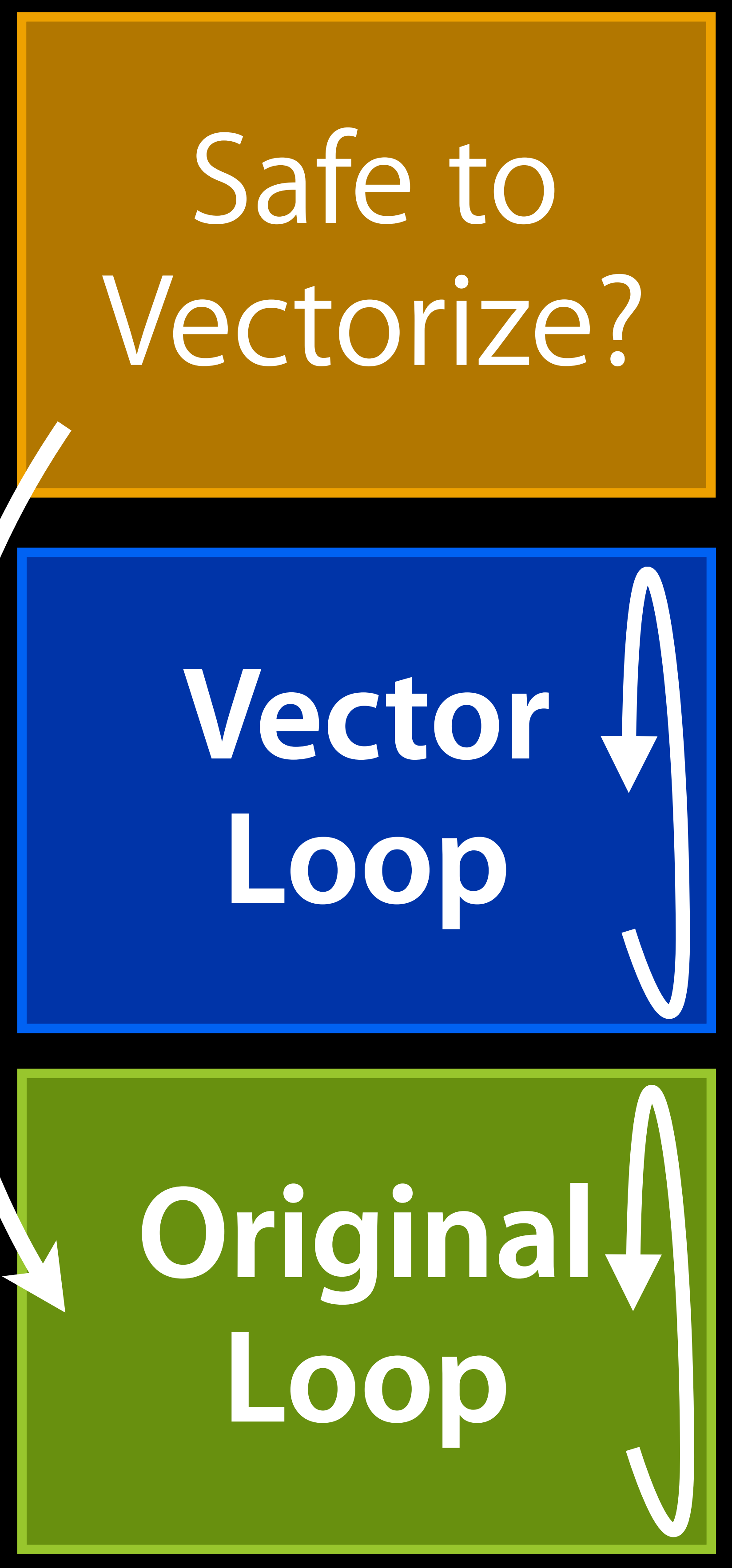
Avoiding Runtime Checks

- Providing more information to the compiler:
  - Pointers point to unique locations
  - Specify number of iterations
- Not practical for most loops

```
void addFP(int N, float *in1,
                  float *in2,
                  float *out) {

  for (i = 0 ; i <   N; i++)
    out[i] = in1[i] + in2[i];

}
```



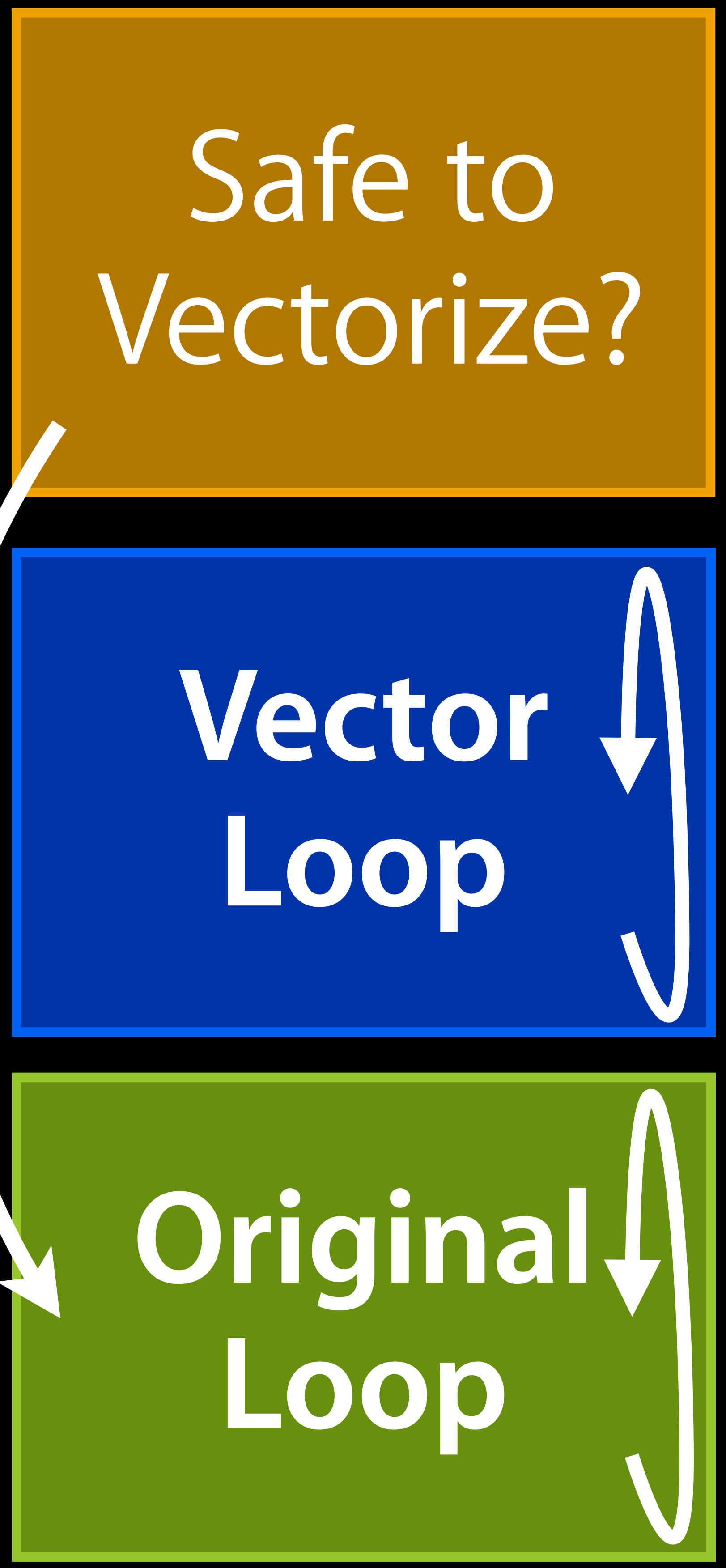
Avoiding Runtime Checks

- Providing more information to the compiler:
  - Pointers point to unique locations
  - Specify number of iterations
- Not practical for most loops

```
void addFP(int N, float *in1,
                  float *in2,
                  float *restrict out) {

  for (i = 0 ; i <   N; i++)
    out[i] = in1[i] + in2[i];

}
```



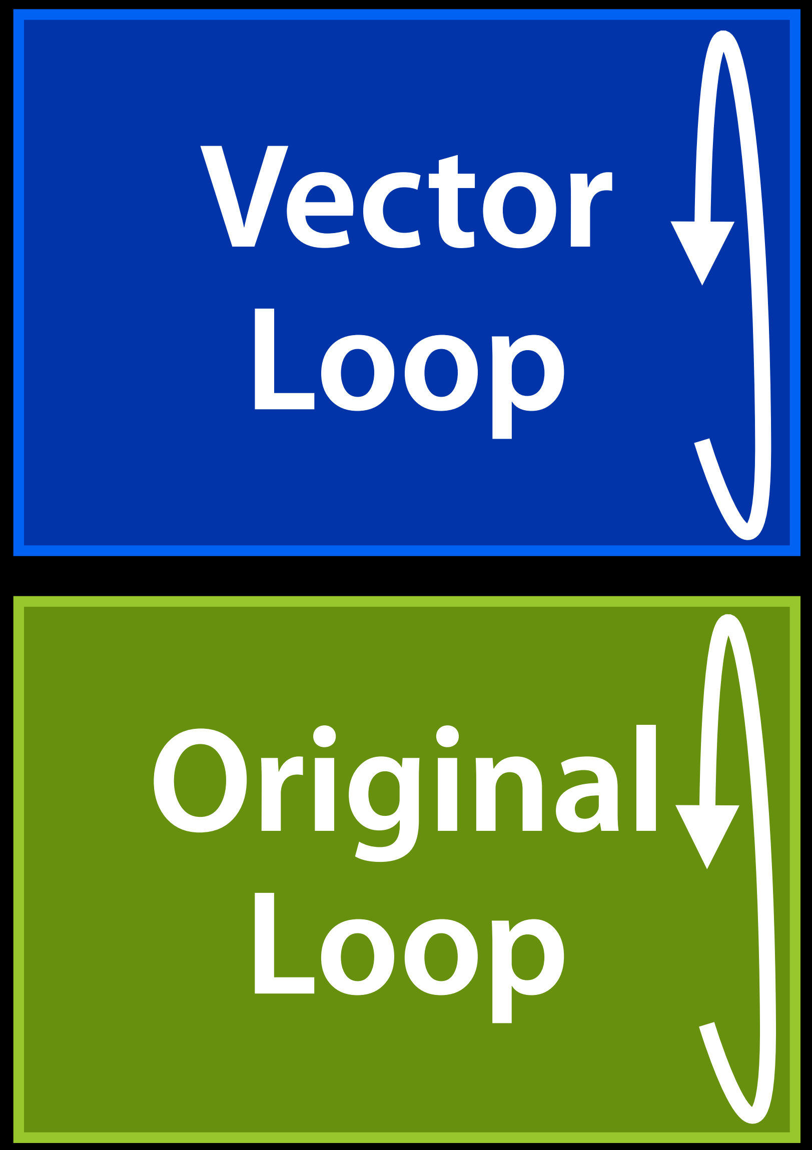
Avoiding Runtime Checks

- Providing more information to the compiler:
    - Pointers point to unique locations
    - Specify number of iterations
- Not practical for most loops

```
void addFP(int N, float *in1,
                  float *in2,
                  float *restrict out) {

  for (i = 0 ; i <   N; i++)
    out[i] = in1[i] + in2[i];

}
```



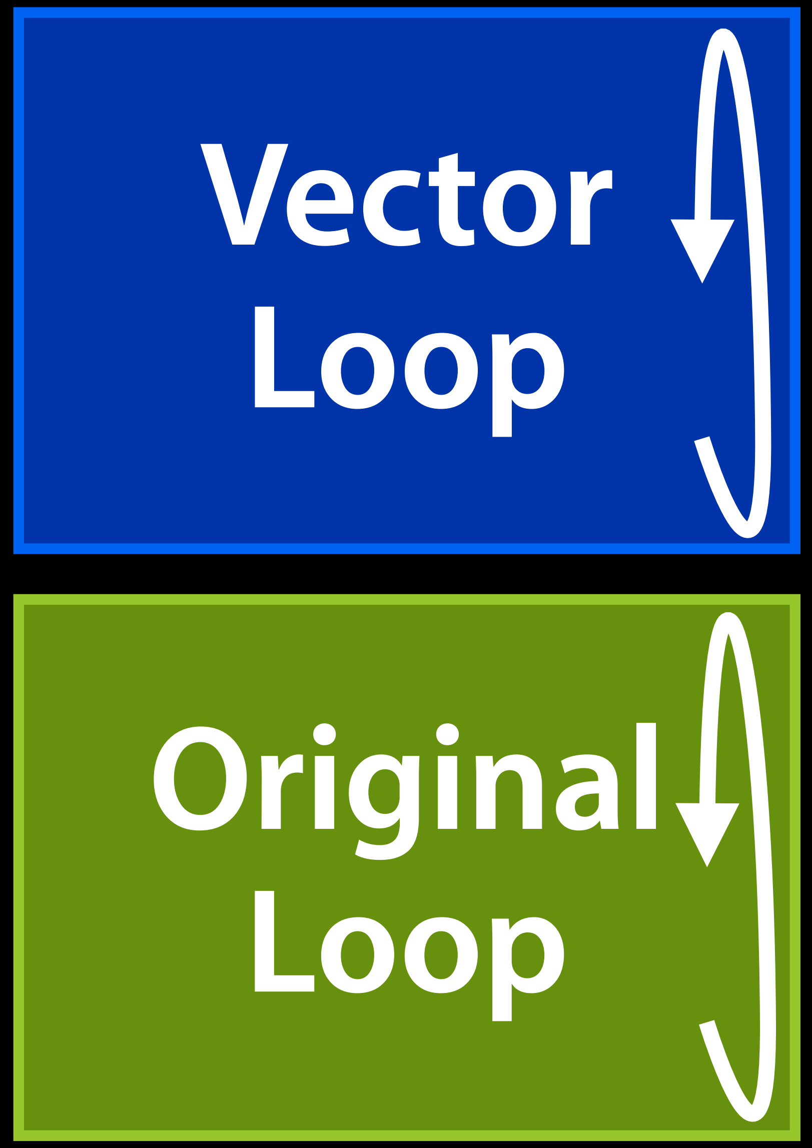
Avoiding Runtime Checks

- Providing more information to the compiler:
    - Pointers point to unique locations
    - Specify number of iterations
- Not practical for most loops

```
void addFP(int N, float *in1,
                  float *in2,
                  float *restrict out) {

  for (i = 0 ; i < 256; i++)
    out[i] = in1[i] + in2[i];

}
```



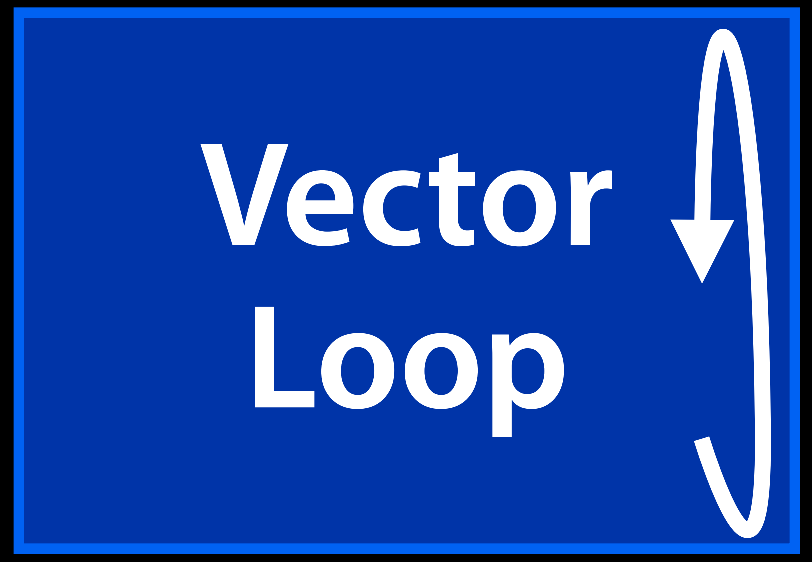
Avoiding Runtime Checks

- Providing more information to the compiler:
  - Pointers point to unique locations
  - Specify number of iterations
- Not practical for most loops

```
void addFP(int N, float *in1,
                  float *in2,
                  float *restrict out) {

  for (i = 0 ; i < 256; i++)
    out[i] = in1[i] + in2[i];

}
```



Vector
Loop

LTO May Help Vectorization

- Compiling the whole program in one piece reveals more information
- More information allows more optimizations
- Vectorize more loops with fewer runtime checks

LTO May Help Vectorization

Loops.c

```
void addFP(int N,
           float *in,
           float *out){
 for (int i = 0; i < N; i++)
    out[i] = in[i] + getK();
}
```

Helper.c

```
int getK() {
 return 4;
}
```

Main.c

```
int main() {
 A = malloc(400);
 B = malloc(400);


 ...
 addFP(100,  A,  B);
}
```

LTO May Help Vectorization

Loops.c

```
void addFP(int N,
           float *in,
           float *out){
 for (int i = 0; i < N; i++)
    out[i] = in[i] + 4;
}
```

Main.c

```
int main() {
 A = malloc(400);
 B = malloc(400);


 ...
 addFP(100,  A,  B);
}
```

LTO May Help Vectorization

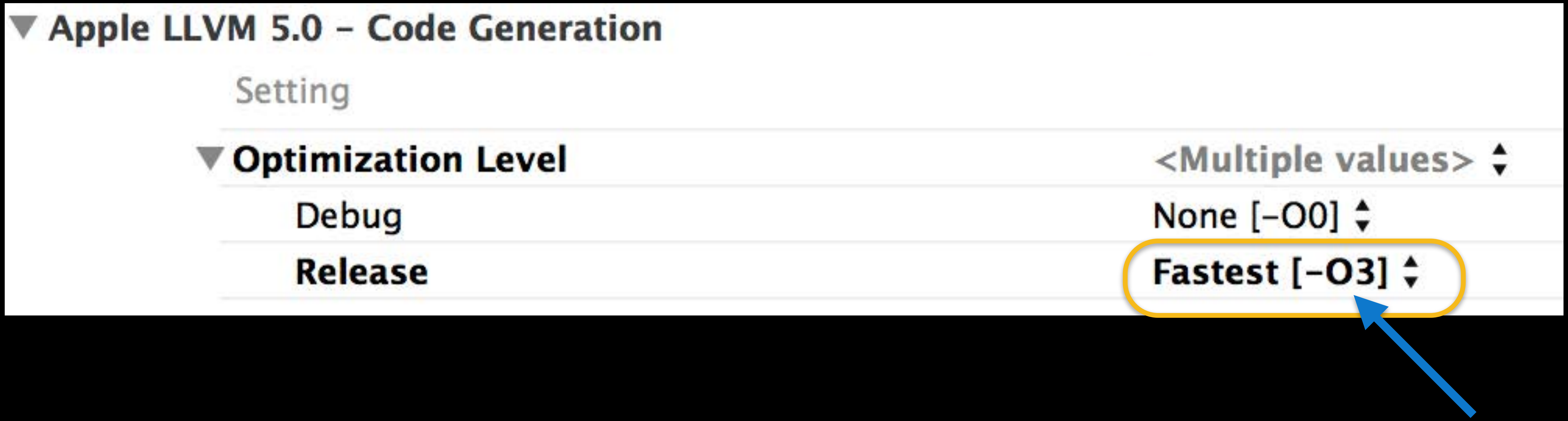
Main.c

```
int main() {
 A = malloc(400);
 B = malloc(400);


 for (int i=0; i<100; i++)
    B[i] = A[i] + 4;
}
```

Vectorization Increases Code Size

- Code size growth is <1% for most applications
- When using -Os the compiler does not vectorize to conserve code size
- Use -O3 to allow vectorization



▼ Apple LLVM 5.0 – Code Generation

Setting	
▼ **Optimization Level**	<Multiple values>
Debug	None [-O0]
Release	**Fastest [-O3]**

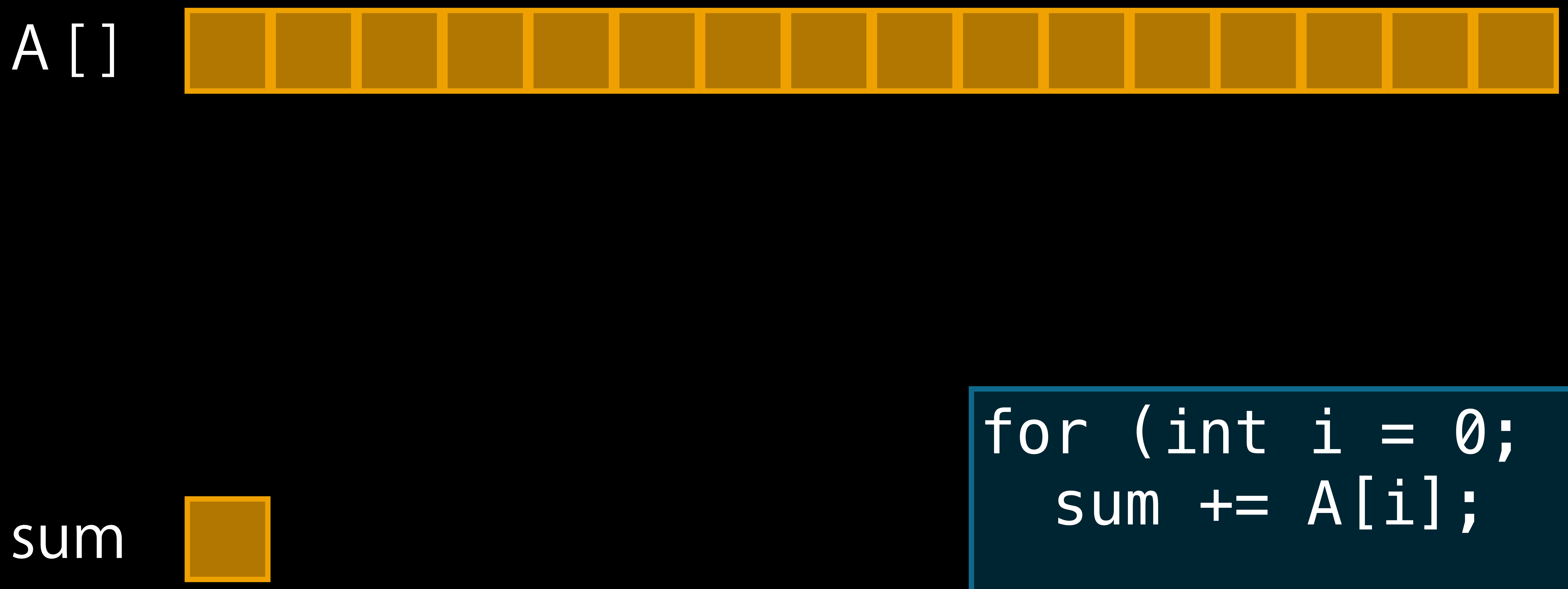
Reduction Loops

Reduction Loops

- Many loops combine multiple elements in the array:
  - Find max element
  - Sum all elements
- Two powerful optimizations

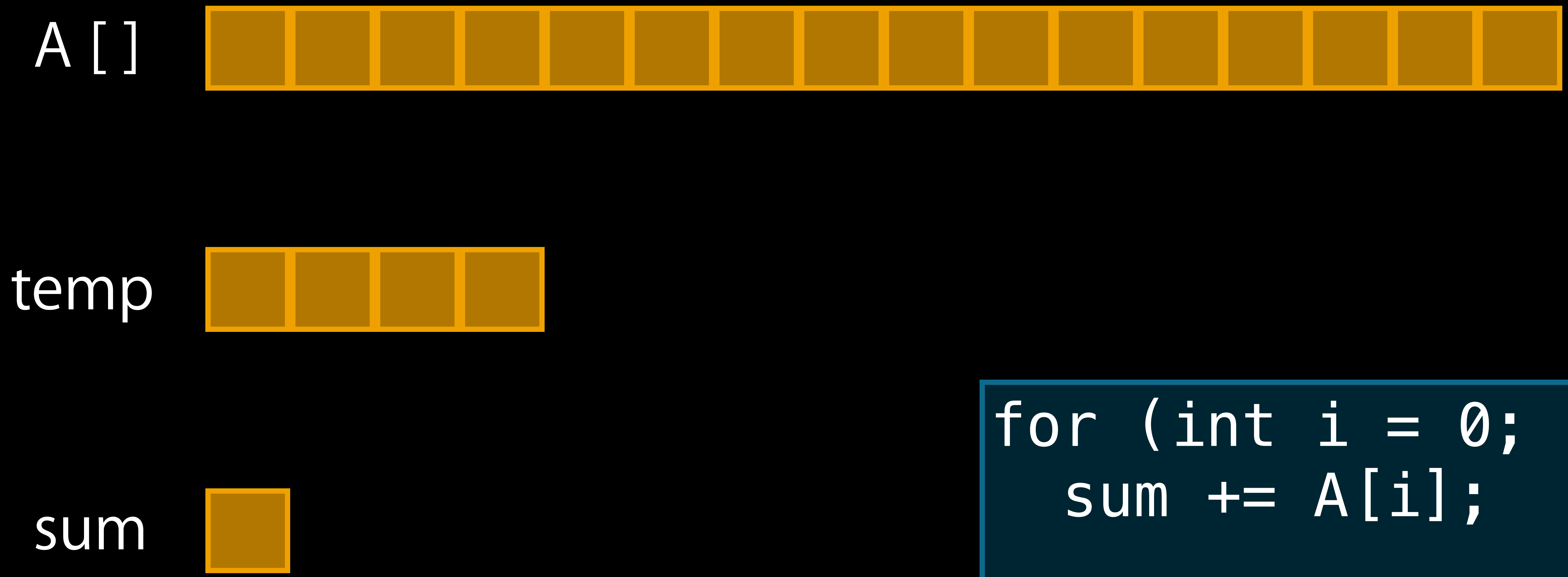
```
for (int i = 0; i < n; i++)
  sum += A[i];
```

Vectorizing Reduction Loops



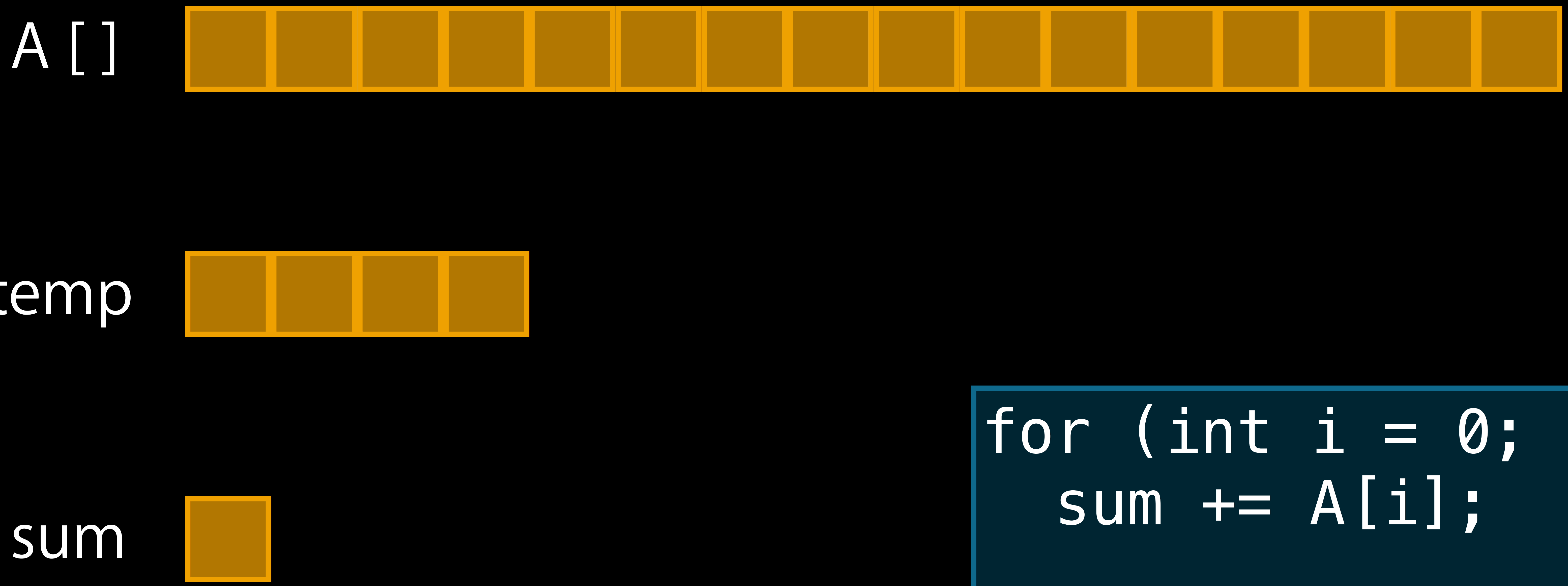
```
for (int i = 0; i < n; i++)
  sum += A[i];
```

Vectorizing Reduction Loops



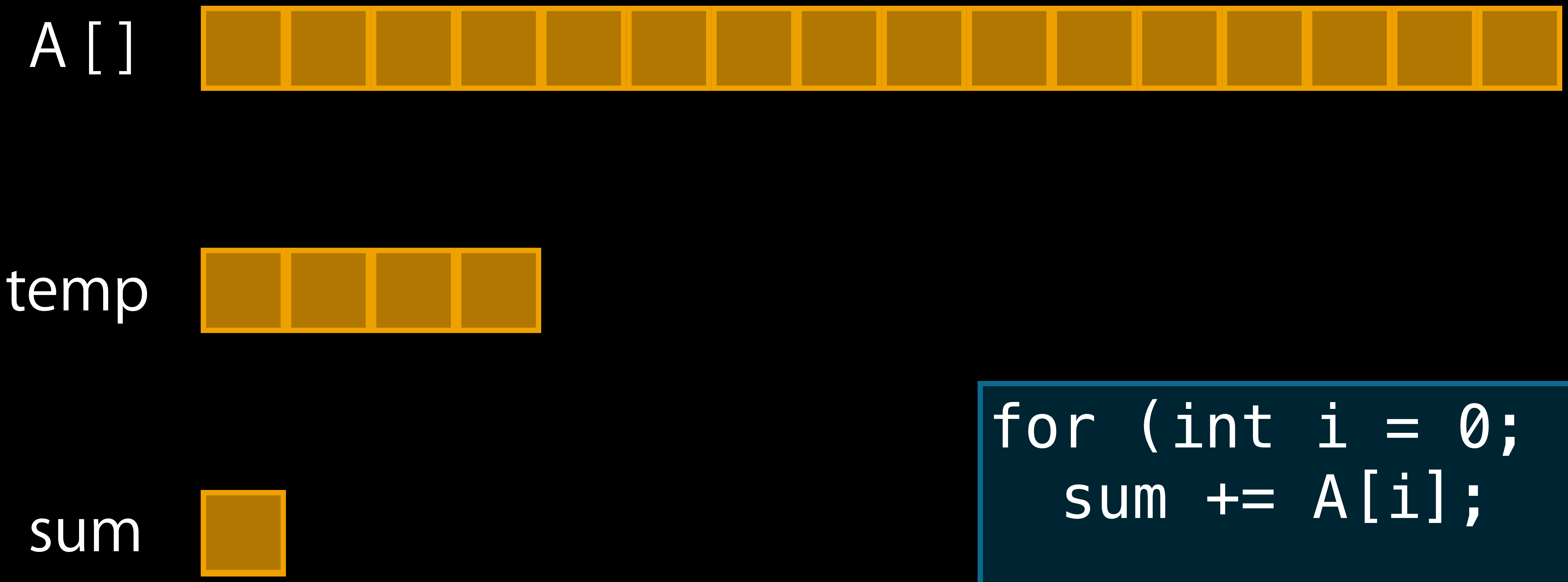
```
for (int i = 0; i < n; i++)
  sum += A[i];
```

Vectorizing Reduction Loops



```
for (int i = 0; i < n; i++)
  sum += A[i];
```

Vectorizing Reduction Loops



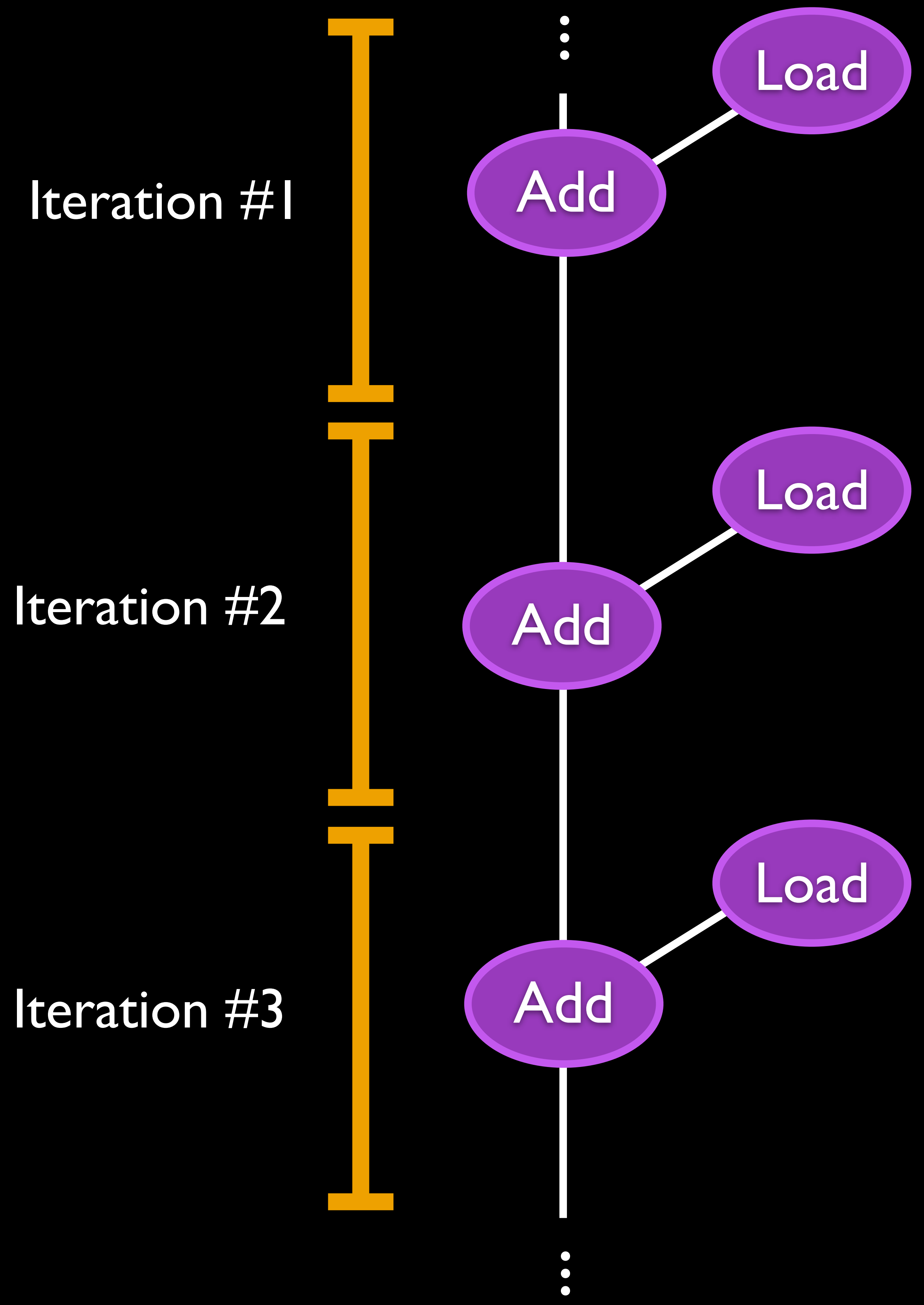
```
for (int i = 0; i < n; i++)
  sum += A[i];
```

Instruction Level Parallelism (ILP)

- Modern CPUs execute independent computations in parallel
  - Called “out-of-order” processors
- LLVM Optimizes for ILP

Out of Order Execution

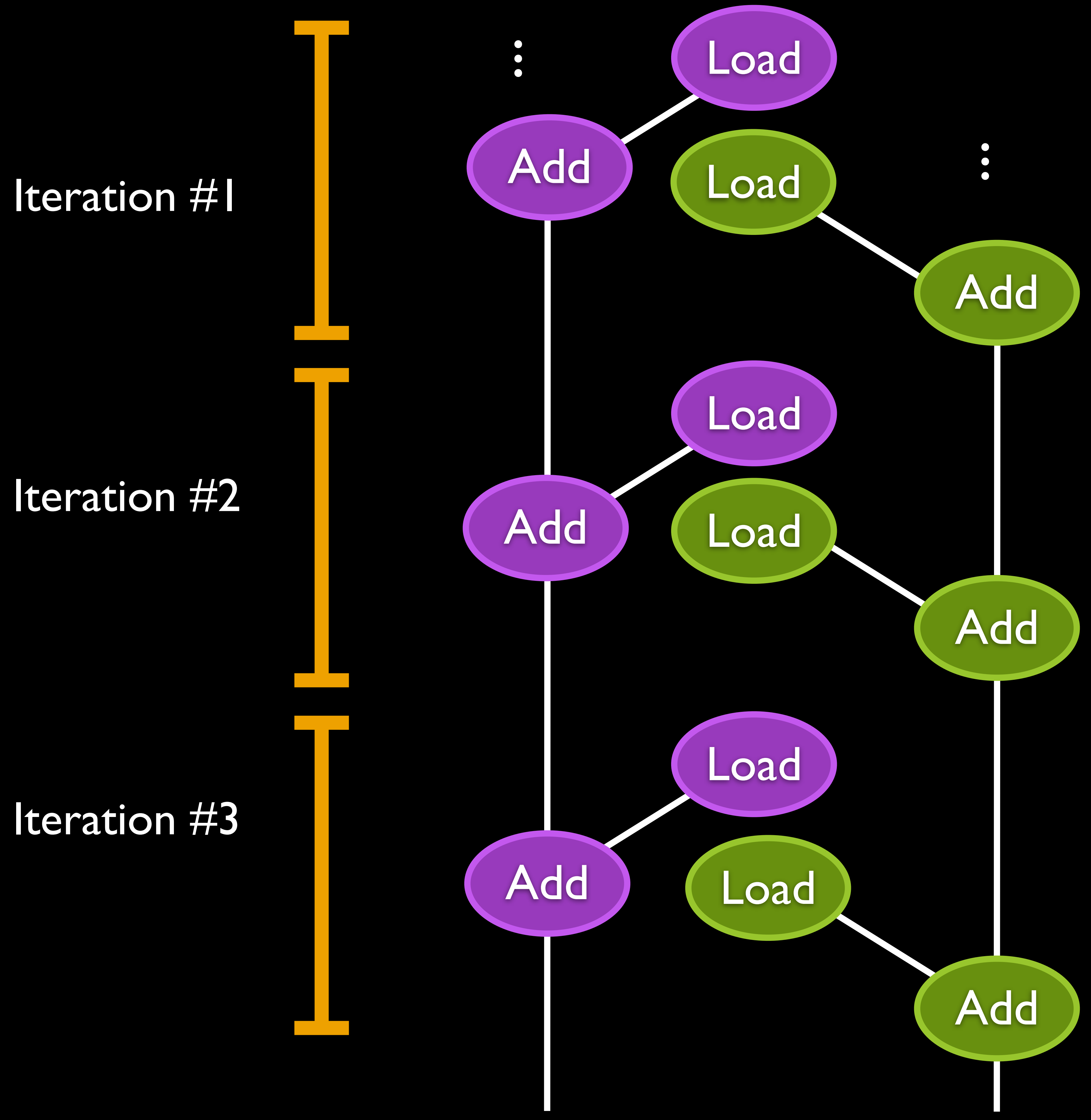
- CPUs "see" multiple iterations of the loop
- Execute only one "add" because we depend on the previous iteration



```
for (int i = 0; i < n; i++)
  sum += A[i];
```

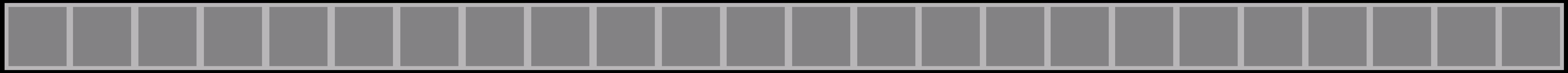
Unrolling

- The compiler unrolls the loop (so you don’t have to)
- Exposes more parallelism for the CPU

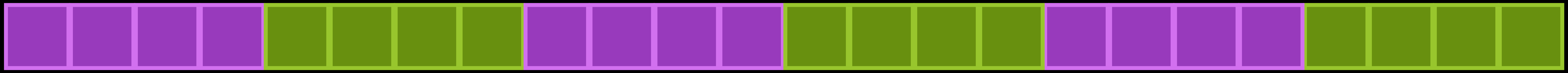


```
for (int i = 0; i < n; i+=2){
 sumA += A[i];
 sumB += A[i+1];
}
```

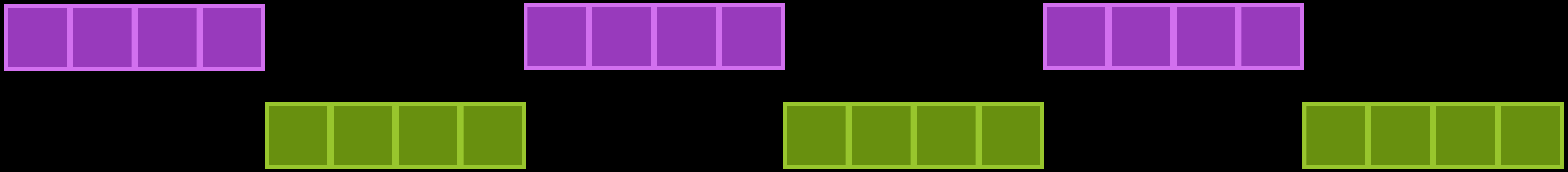
Vector + Unroll



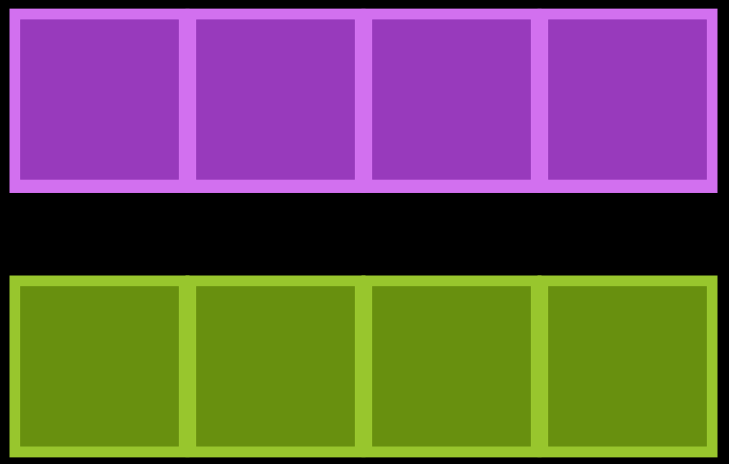
Vector + Unroll



Vector + Unroll

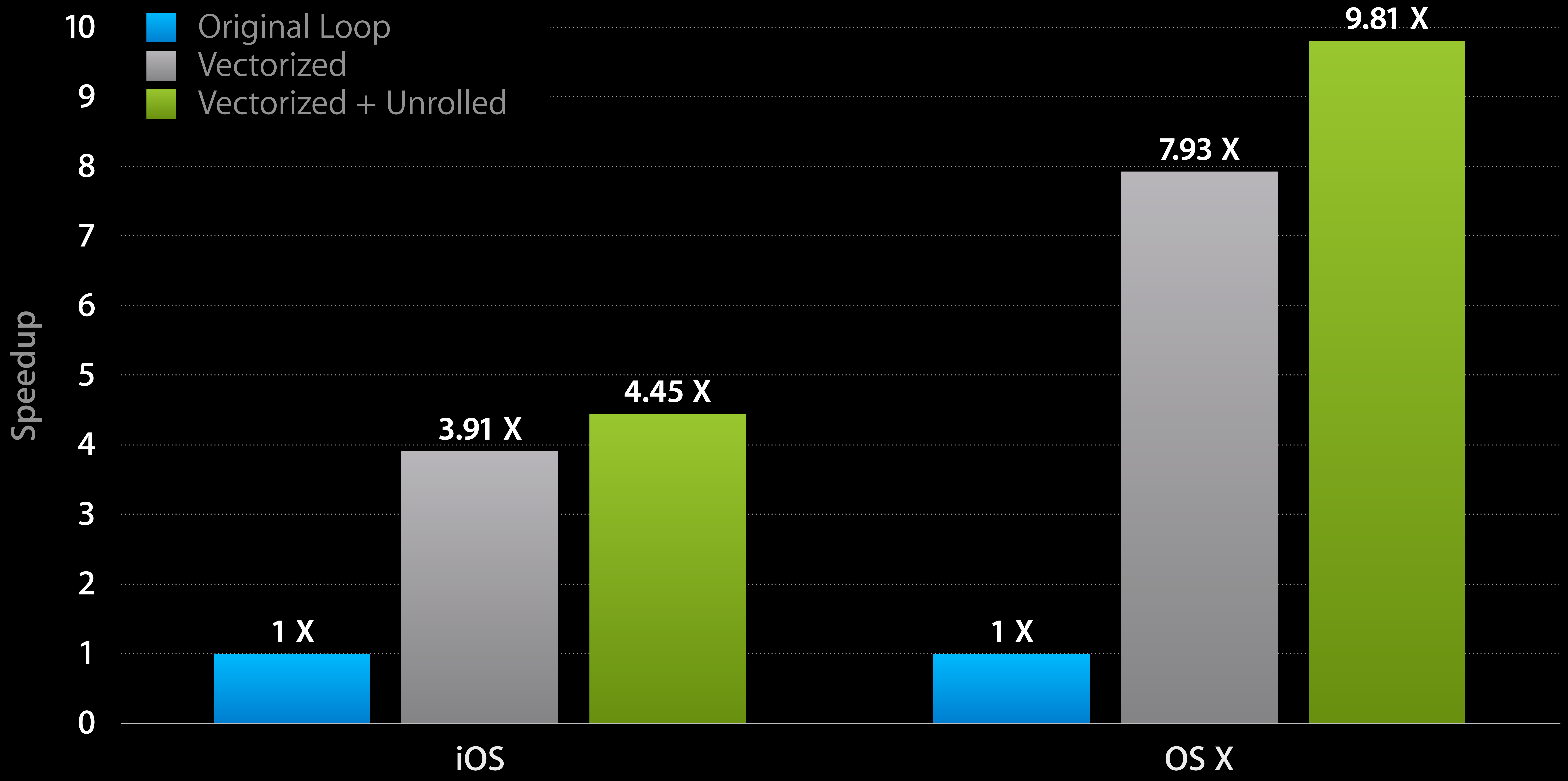


Vector + Unroll



```
L0:
  vld1.32     {d20, d21}, [r2]
  vadd.f32    q8, q8, q10
  add.w       r3, r2, #16
  subs.w      r12, r12, #8
  add.w       r2, r2, #32
  vld1.32     {d20, d21}, [r3]
  vadd.f32    q9, q9, q10
  bne         L0
```

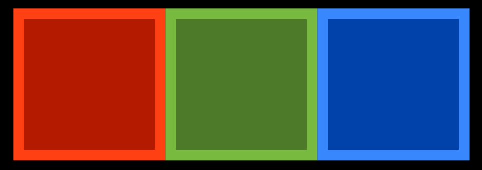
Vectorized Unrolled "sum" Loop

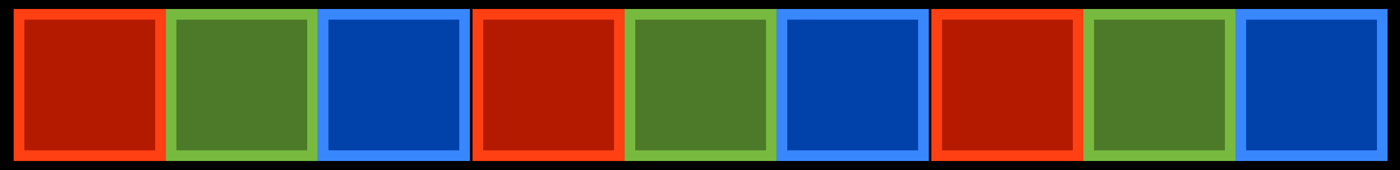
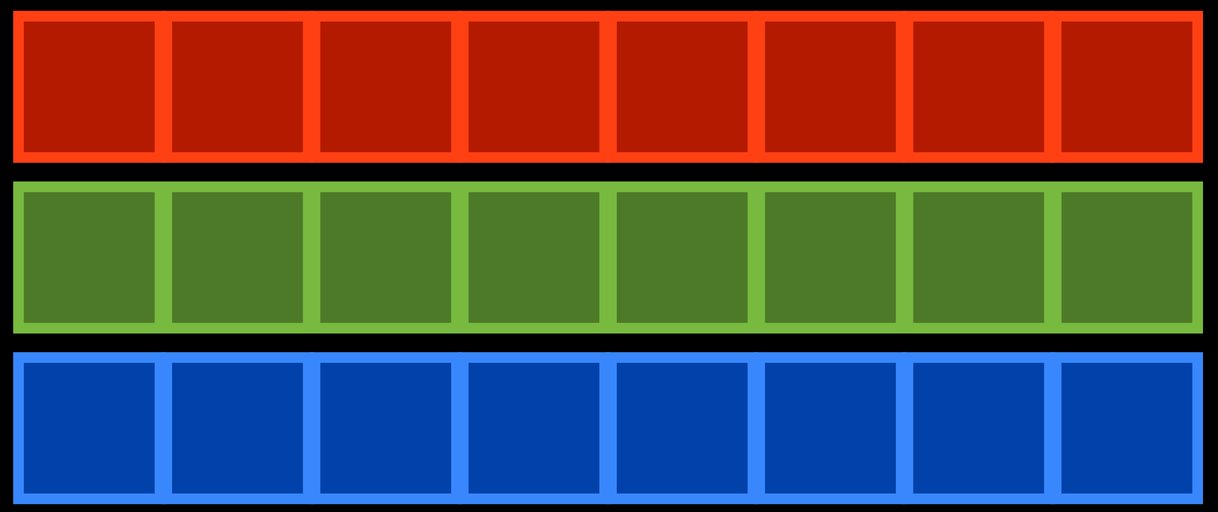


Optimizing Floating Point Code

- Requires re-ordering calculations
- Use -ffast-math to enable vectorization

Data Layout Effects

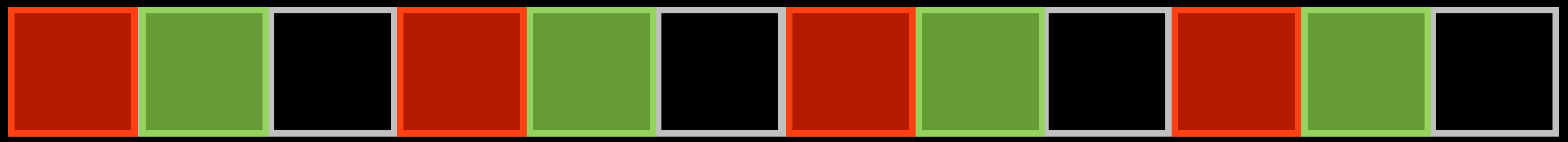
- This is a pixel 

- Two ways to store multiple pixels:
  - Array of Structs (AoS) 
  - Struct of Arrays (SoA) 

Array of Structs (AoS) Layout

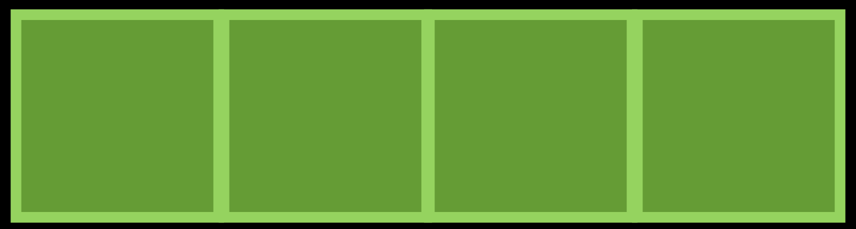
- AoS requires non-consecutive accesses
- Gathering memory is expensive
- LLVM detects that it is not profitable to vectorize

```
for (i = 0; i < N; i++) {
    A[i].r += A[i].g;
}
```

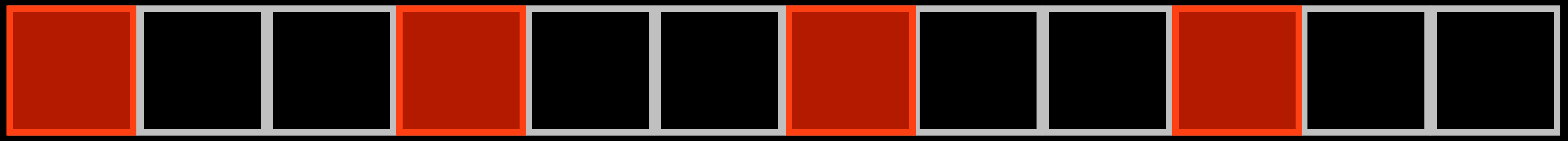


Array of Structs (AoS) Layout

- AoS requires non-consecutive accesses
- Gathering memory is expensive
- LLVM detects that it is not profitable to vectorize



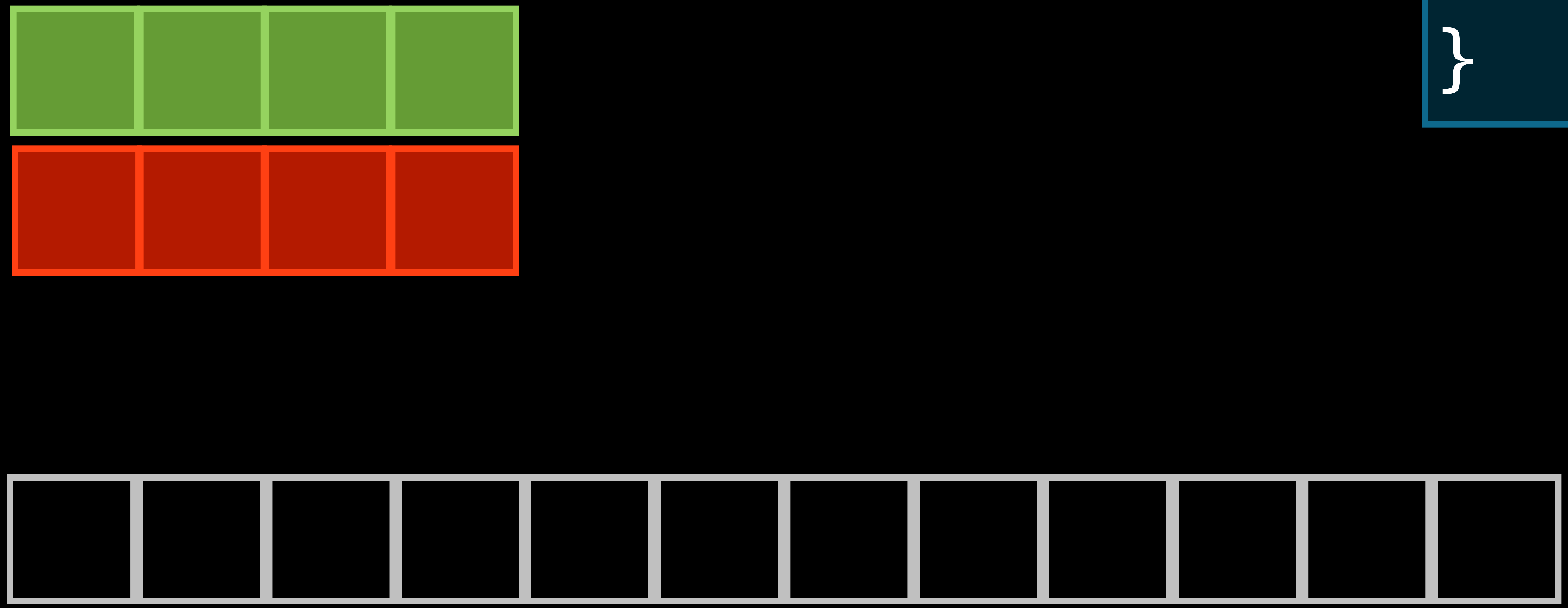
```
for (i = 0; i < N; i++) {
    A[i].r += A[i].g;
}
```



Array of Structs (AoS) Layout

- AoS requires non-consecutive accesses
- Gathering memory is expensive
- LLVM detects that it is not profitable to vectorize

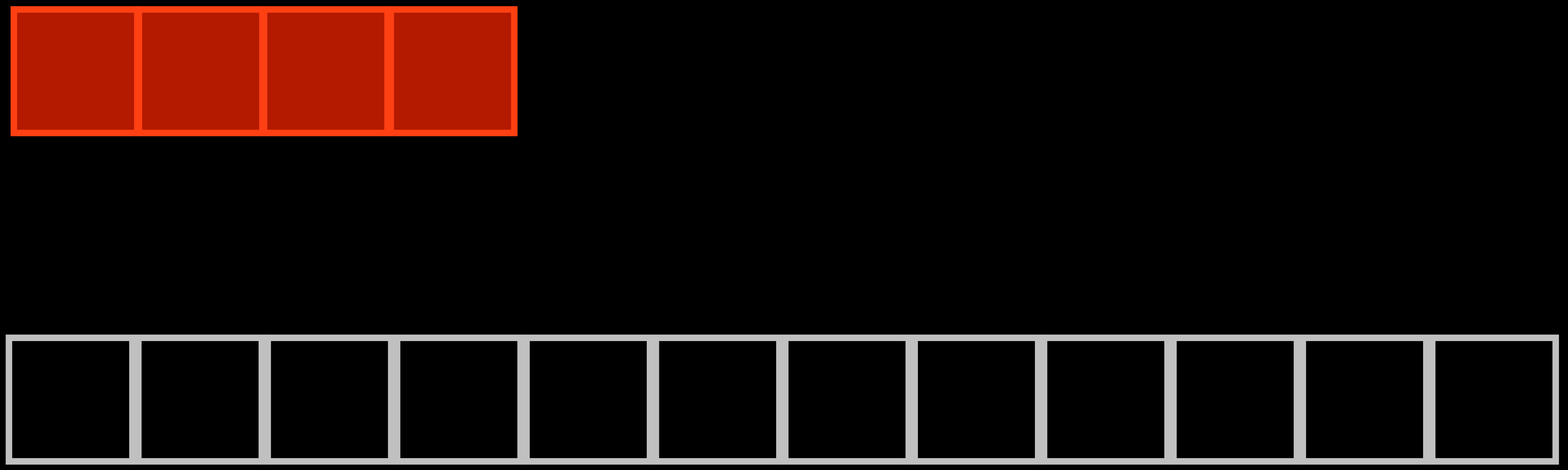
```
for (i = 0; i < N; i++) {
    A[i].r += A[i].g;
}
```



Array of Structs (AoS) Layout

- AoS requires non-consecutive accesses
- Gathering memory is expensive
- LLVM detects that it is not profitable to vectorize

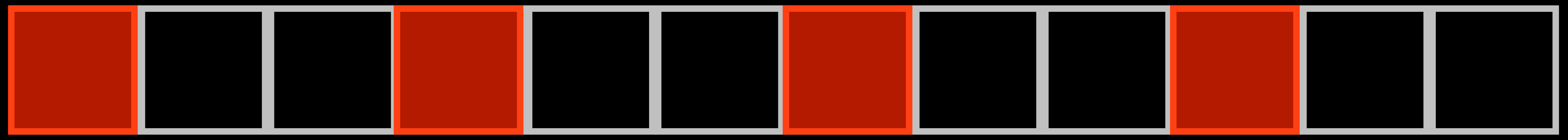
```
for (i = 0; i < N; i++) {
    A[i].r += A[i].g;
}
```



Array of Structs (AoS) Layout

- AoS requires non-consecutive accesses
- Gathering memory is expensive
- LLVM detects that it is not profitable to vectorize

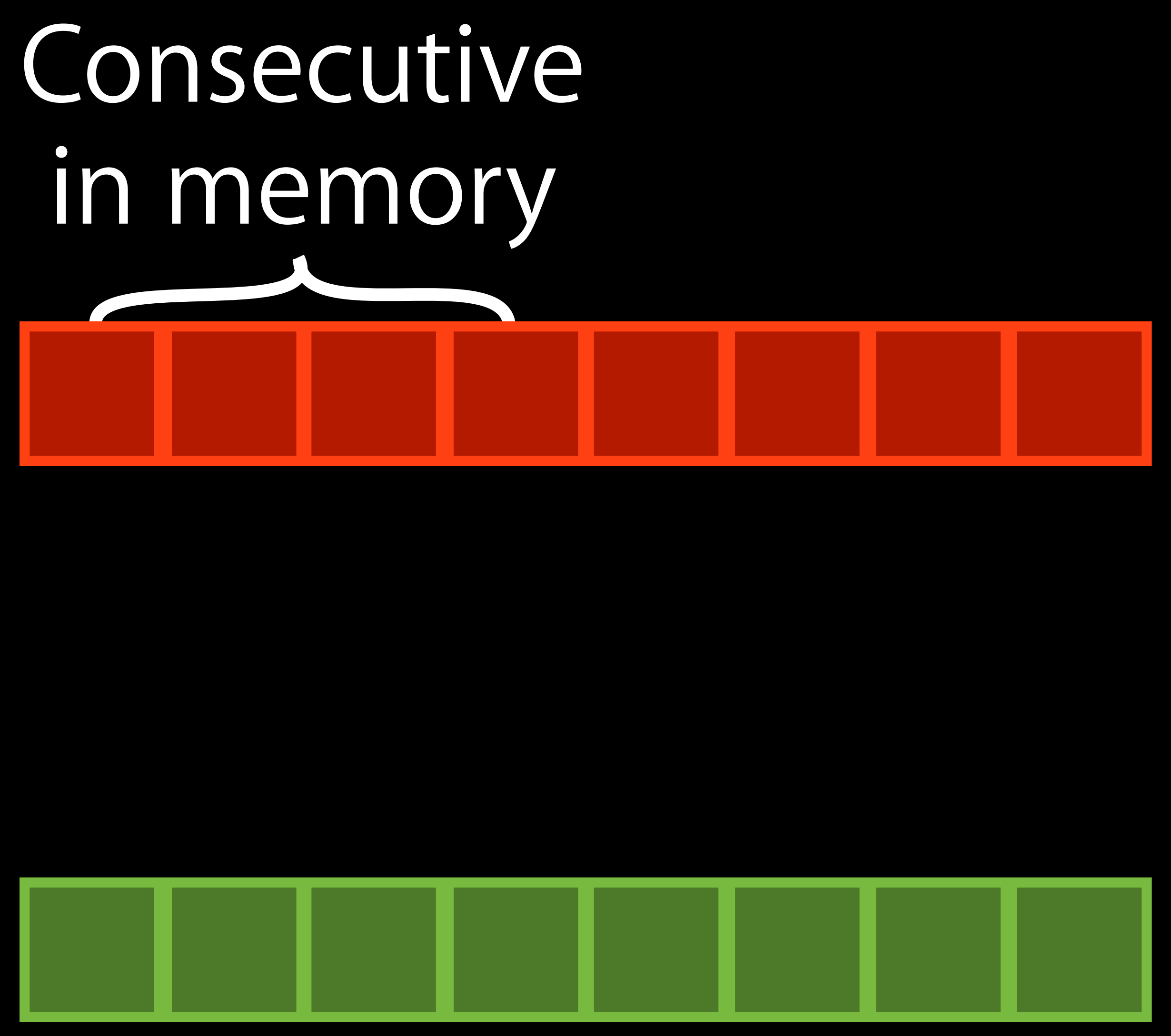
```
for (i = 0; i < N; i++) {
    A[i].r += A[i].g;
}
```



Struct of Arrays (SoA) Layout

- SoA allows fast consecutive memory access
- Prefer consecutive access of arrays with primitive types
- The compiler will not do it for you

```
for (i = 0; i < N; i++) {
    R[i] += G[i];
}
```

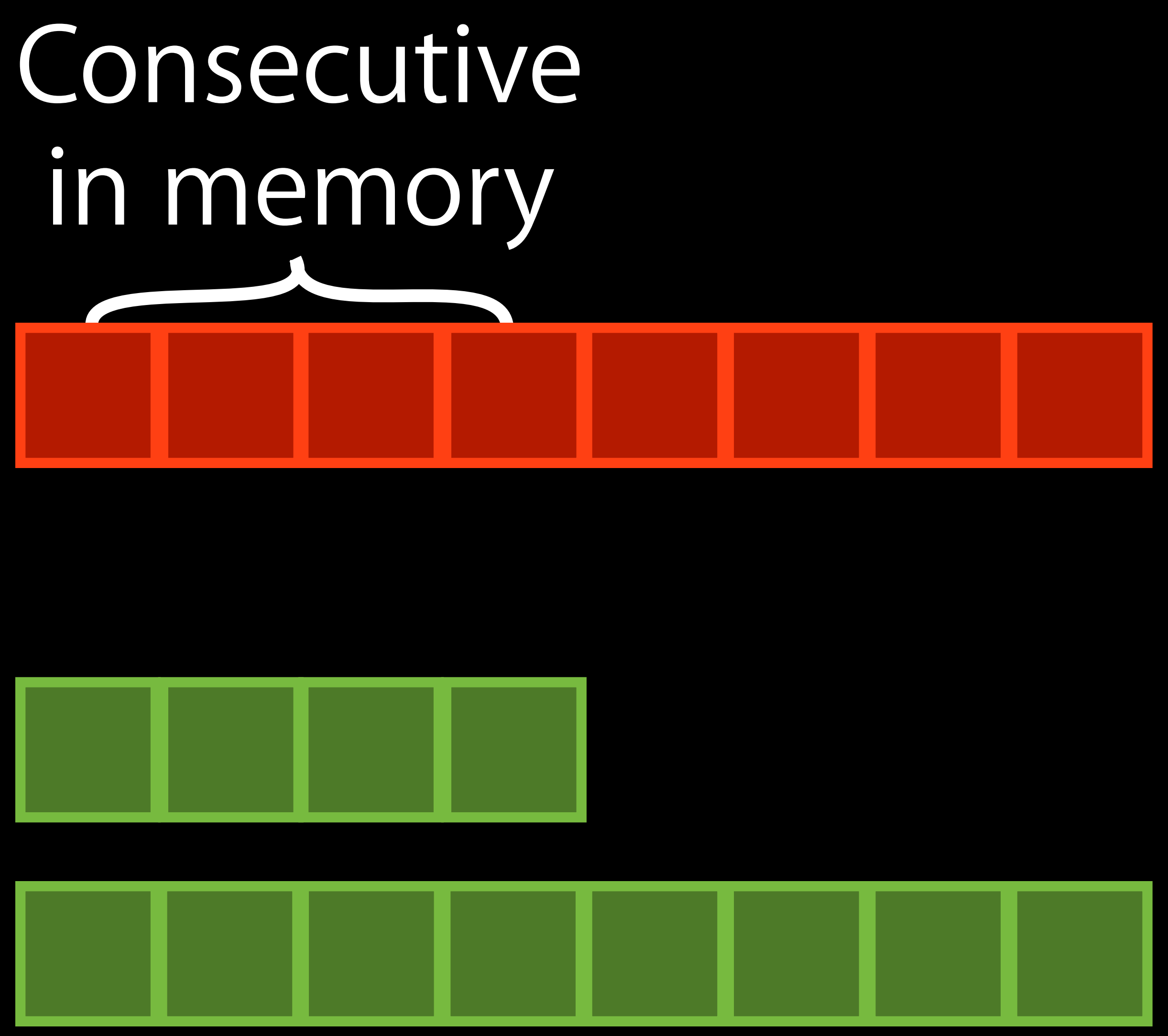


Consecutive
in memory

Struct of Arrays (SoA) Layout

- SoA allows fast consecutive memory access
- Prefer consecutive access of arrays with primitive types
- The compiler will not do it for you

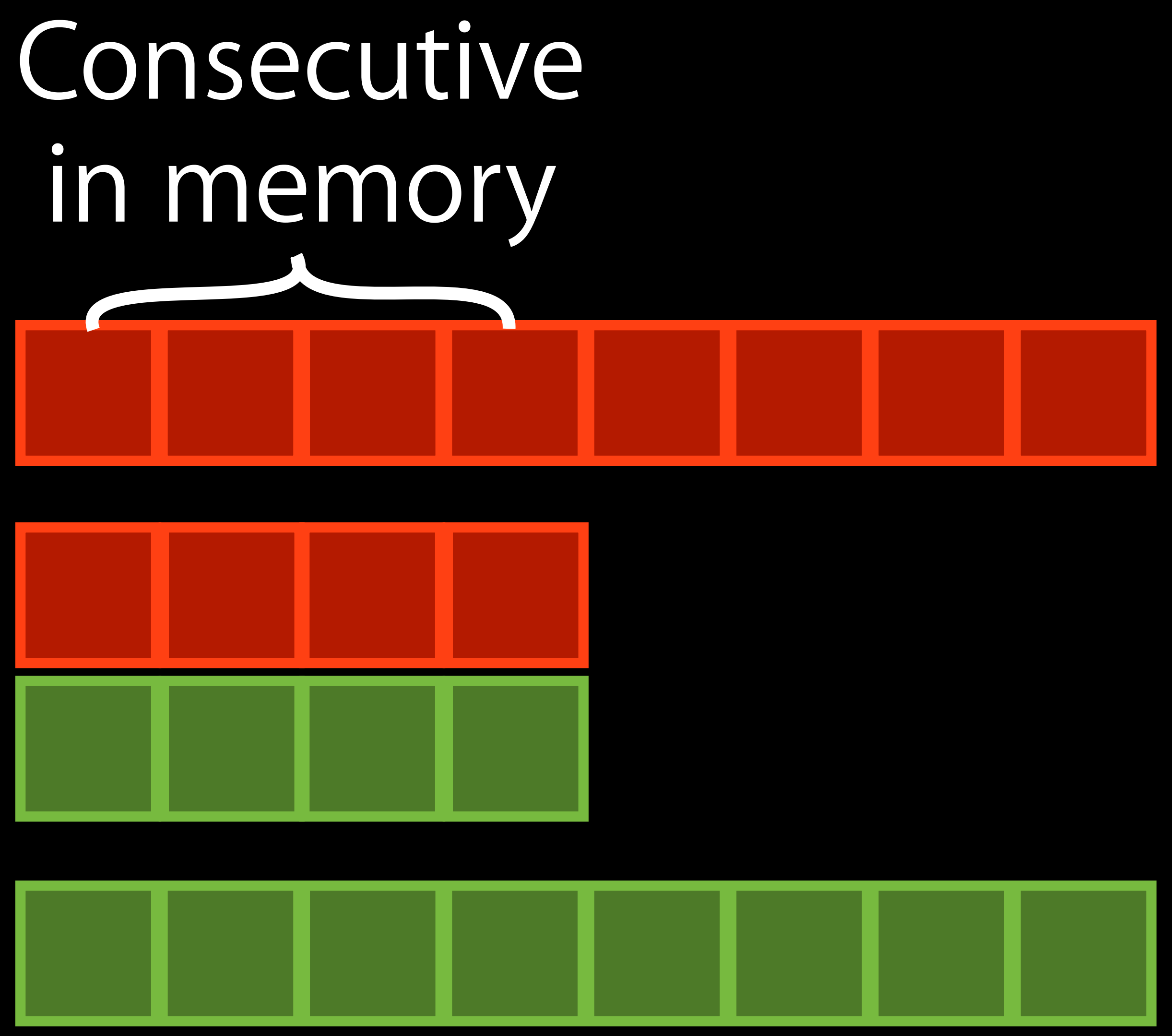
```
for (i = 0; i < N; i++) {
    R[i] += G[i];
}
```



Struct of Arrays (SoA) Layout

- SoA allows fast consecutive memory access
- Prefer consecutive access of arrays with primitive types
- The compiler will not do it for you

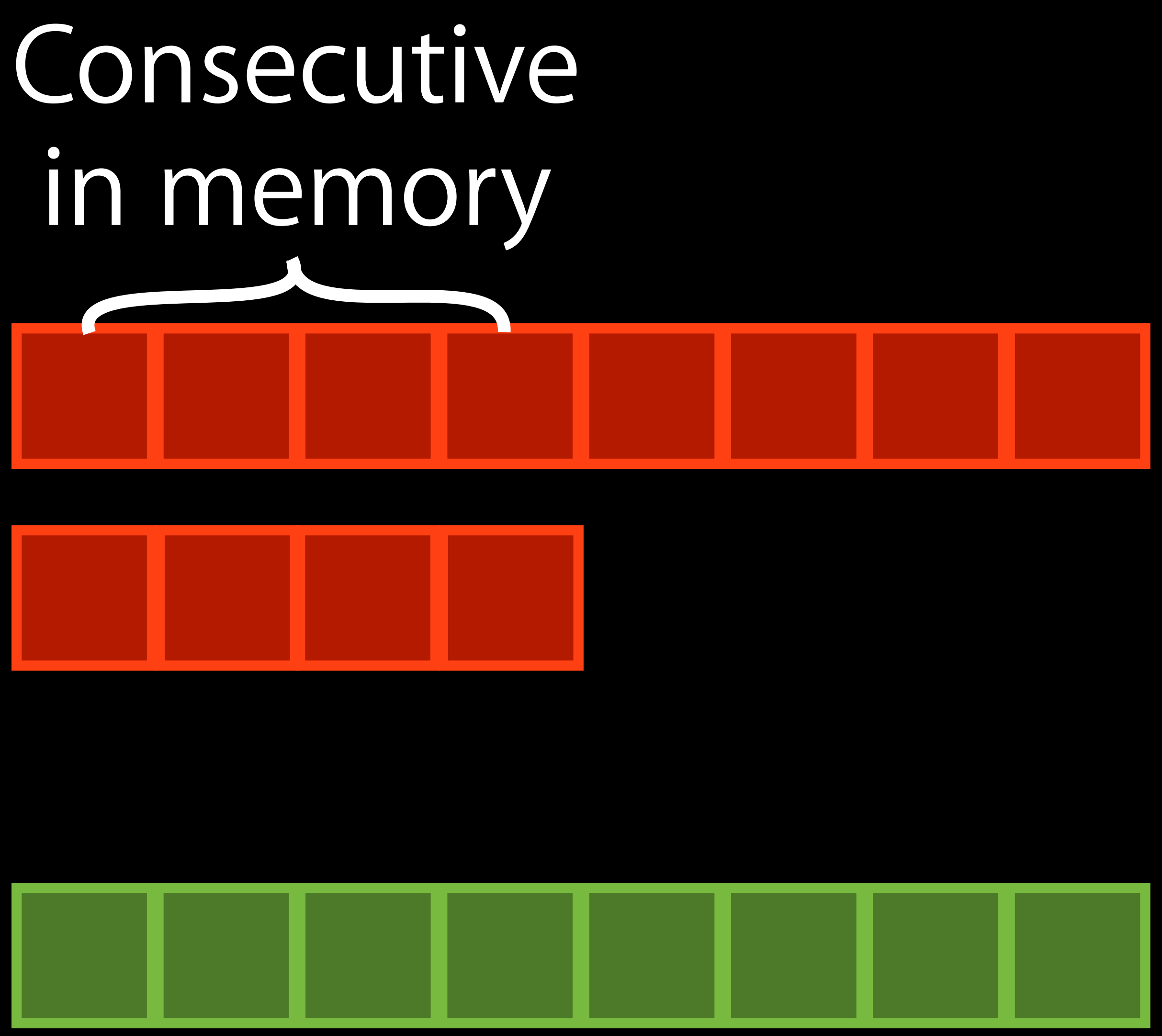
```
for (i = 0; i < N; i++) {
    R[i] += G[i];
}
```



Struct of Arrays (SoA) Layout

- SoA allows fast consecutive memory access
- Prefer consecutive access of arrays with primitive types
- The compiler will not do it for you

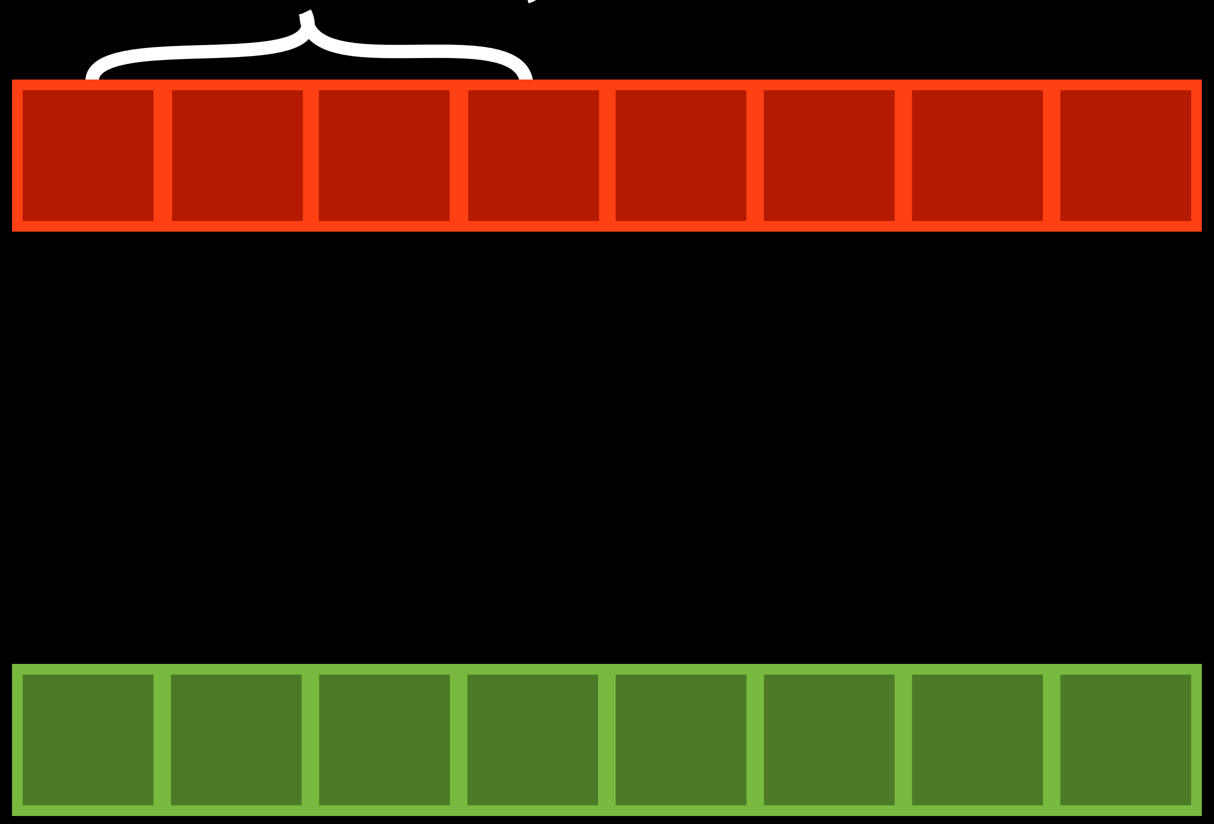
```
for (i = 0; i < N; i++) {
    R[i] += G[i];
}
```



Struct of Arrays (SoA) Layout

- SoA allows fast consecutive memory access
- Prefer consecutive access of arrays with primitive types
- The compiler will not do it for you

```
for (i = 0; i < N; i++) {
    R[i] += G[i];
}
```



Signed vs. Unsigned Indices

```
int foo(int *A, int start, int end) {
  for (int i = start; i < end ; ++i)
    A[i] += 3;
}
```

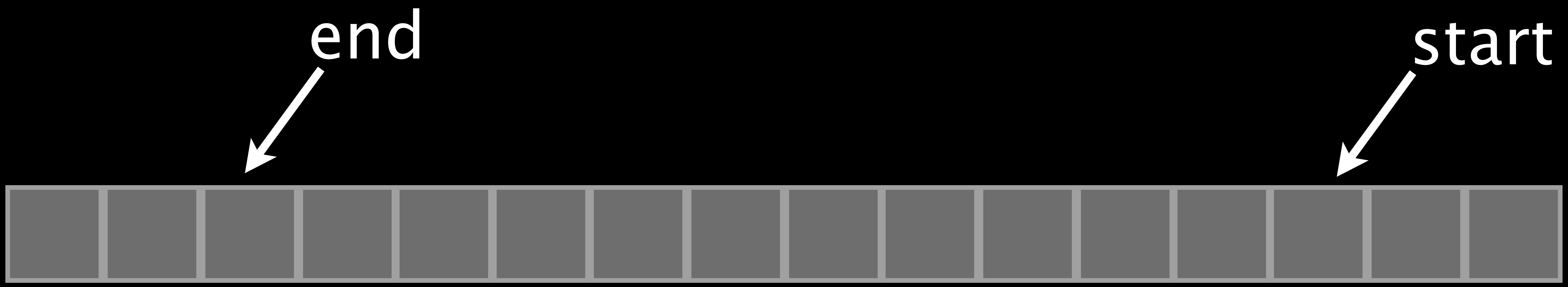
✓

```
int foo2(int *A, unsigned start, unsigned end) {
  for (unsigned i = start; i < end ; ++i)
    A[i] += 3;
}
```

✗

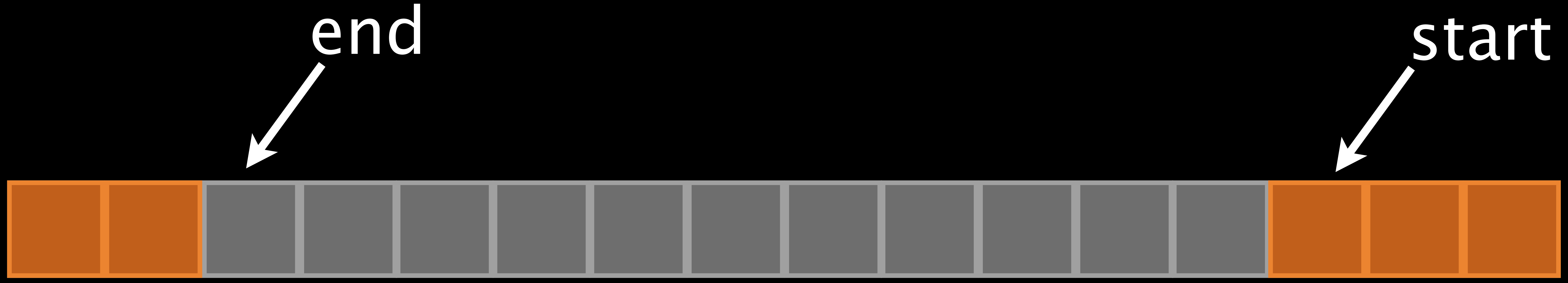
Unsigned Indices

- The C language specifies that unsigned integers are allowed to wrap
- The compiler assumes that you want your indices to wrap
- Use signed indices, or **`size_t`**



Unsigned Indices

- The C language specifies that unsigned integers are allowed to wrap
- The compiler assumes that you want your indices to wrap
- Use signed indices, or **`size_t`**



```
int foo(int *A, size_t start, size_t end) {
  for (size_t i = start; i < end ; ++i)
    A[i] += 3;
}
```

✓

Putting it All Together

Optimizing Your Code Using LLVM

- Write simple code
- Select the right optimization flags

Optimizations We Talked About

- -O3
- Fast math
- Vectorizer
- Strict-aliasing
- LTO

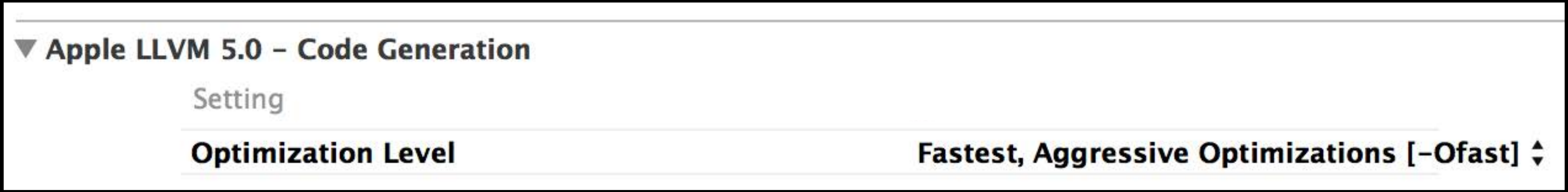
Optimizations We Talked About

- -Ofast
- LTO

-Ofast

NEW

- New optimization level to enable:
  - -O3
  - vectorization
  - strict aliasing
  - fast math



▼ Apple LLVM 5.0 – Code Generation

Setting	
Optimization Level	Fastest, Aggressive Optimizations [–Ofast]

More Information

Dave DeLong
Developer Tools Evangelist
delong@apple.com

LLVM Project
Open-Source LLVM Project Home
http://llvm.org

Apple Developer Forums
http://devforums.apple.com

Labs

Objective-C and LLVM Lab	Tools Lab C Thursday 2:00PM

WWDC2013