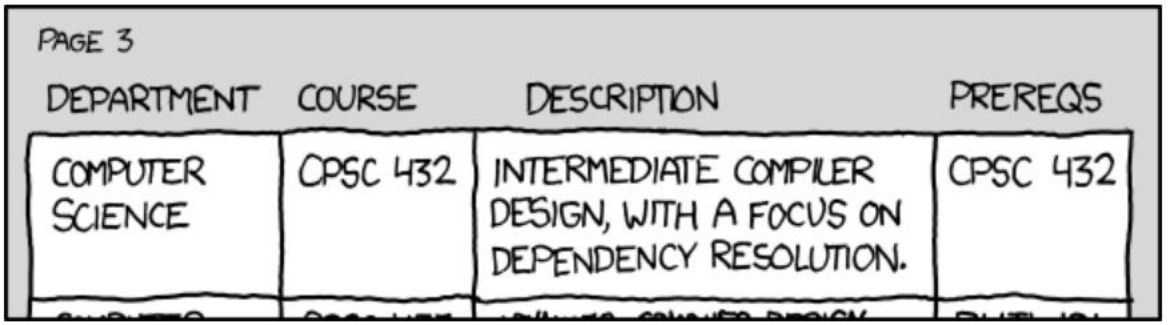
Compilers - Chapter 8:

Loop scheduling optimisations

Part 1: Why mess with the order of loop execution?

- Lecturer:
  - Paul Kelly (p.kelly@imperial.ac.uk)



(It turns out that this cartoon is good for almost any compilers topic, definitely this one) https://xkcd.com/754/

“Restructuring” compilers

- The optimisations we have studied so far reduce the number of instructions that need to be executed at runtime
  - This is fundamentally a good idea!
- But sometimes we can get a performance improvement by thinking about the order in which loops are executed
- Why might that be?

"Restructuring" compilers

- The optimisations we have studied so far reduce the number of instructions that need to be executed at runtime
  - This is fundamentally a good idea!
- But sometimes we can get a performance improvement by thinking about the order in which loops are executed
- Why might that be?
  - We might be able to use vector instructions
    - So different iterations of a loop are being executed at the same time

“Restructuring” and “parallelizing” compilers

- The optimisations we have studied so far reduce the number of instructions that need to be executed at runtime
  - This is fundamentally a good idea!
- But sometimes we can get a performance improvement by thinking about the order in which loops are executed
- Why might that be?
  - We might be able to use vector instructions
    - So different iterations of a loop are being executed at the same time
  - We might be able to use multiple cores
    - So different iterations of a loop might be assigned to different threads running on different CPUs

“Restructuring” and “parallelizing” compilers

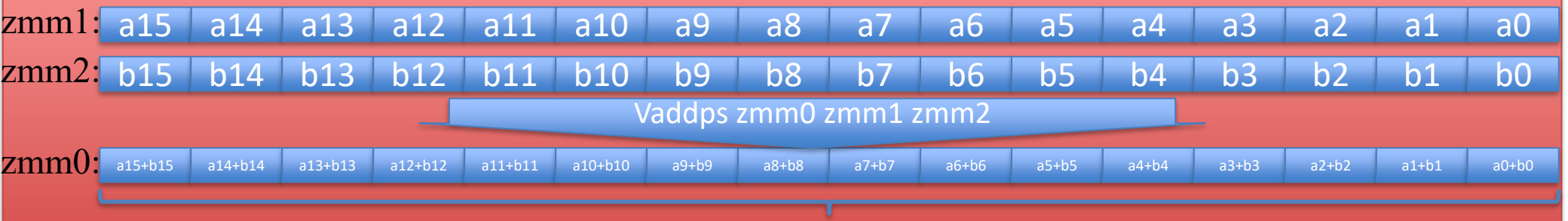
- The optimisations we have studied so far reduce the number of instructions that need to be executed at runtime
  - This is fundamentally a good idea!
- But sometimes we can get a performance improvement by thinking about the order in which loops are executed
- Why might that be?
  - We might be able to use vector instructions
    - So different iterations of a loop are being executed at the same time
  - We might be able to use multiple cores
    - So different iterations of a loop might be assigned to different threads running on different CPUs
  - We might be able to improve how the cache is used
    - We will come to this later!

Vector instruction set extensions

- Example: Intel's AVX512
- Extended registers ZMM0-ZMM31, 512 bits wide
  - Can be used to store 8 doubles, 16 floats, 32 shorts, 64 bytes
  - So instructions are executed in parallel in 64,32,16 or 8 "lanes"

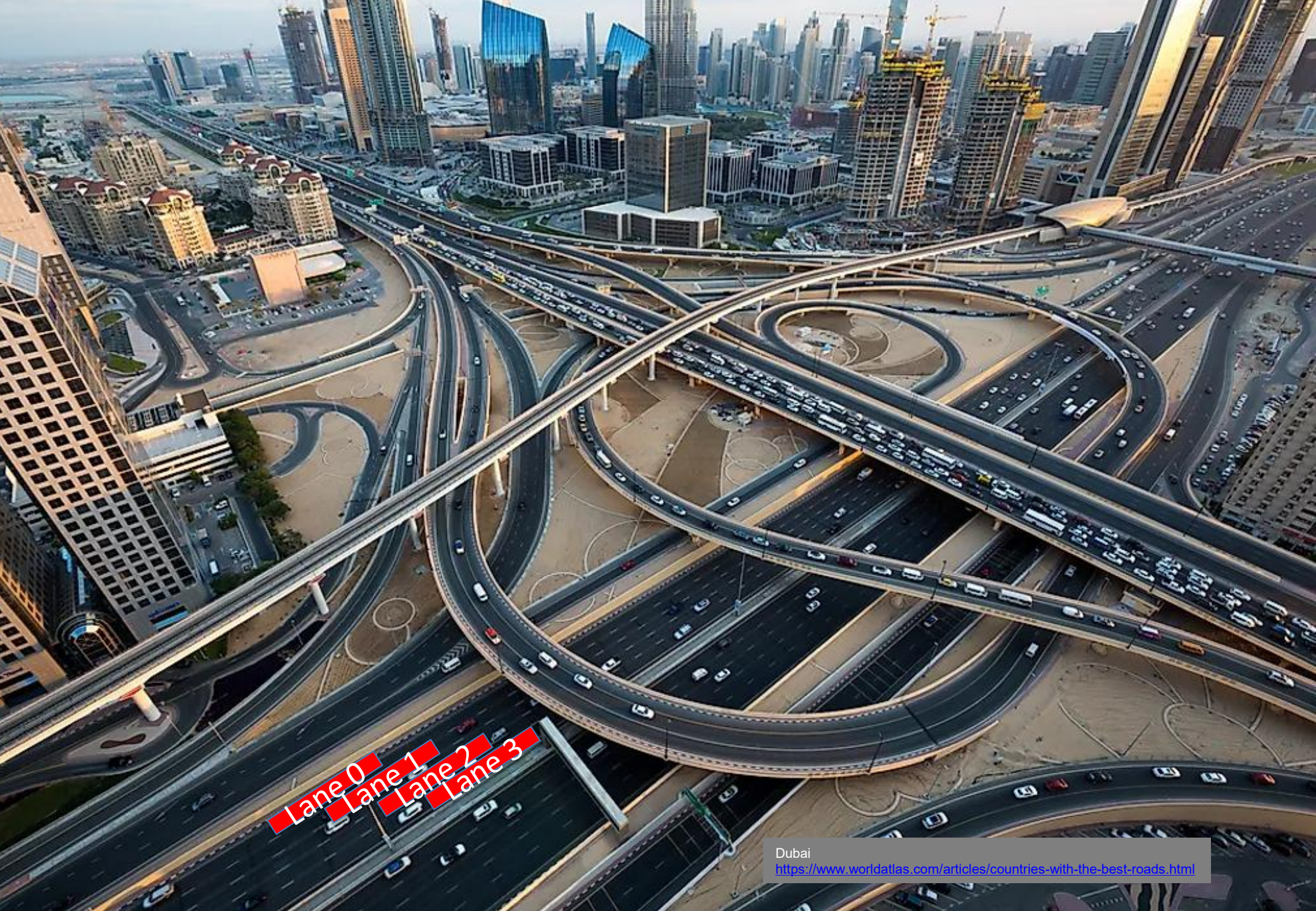
Vector instruction set extensions

- Example: Intel's AVX512
- Extended registers ZMM0-ZMM31, 512 bits wide
  - Can be used to store 8 doubles, 16 floats, 32 shorts, 64 bytes
- Example: `vaddps zmm0 zmm1 zmm2`
  - "Add Packed Single Precision Floating-Point Values"



In *one instruction* we add 16 32-bit floating point values from zmm1 and 16 32-bit values from zmm2

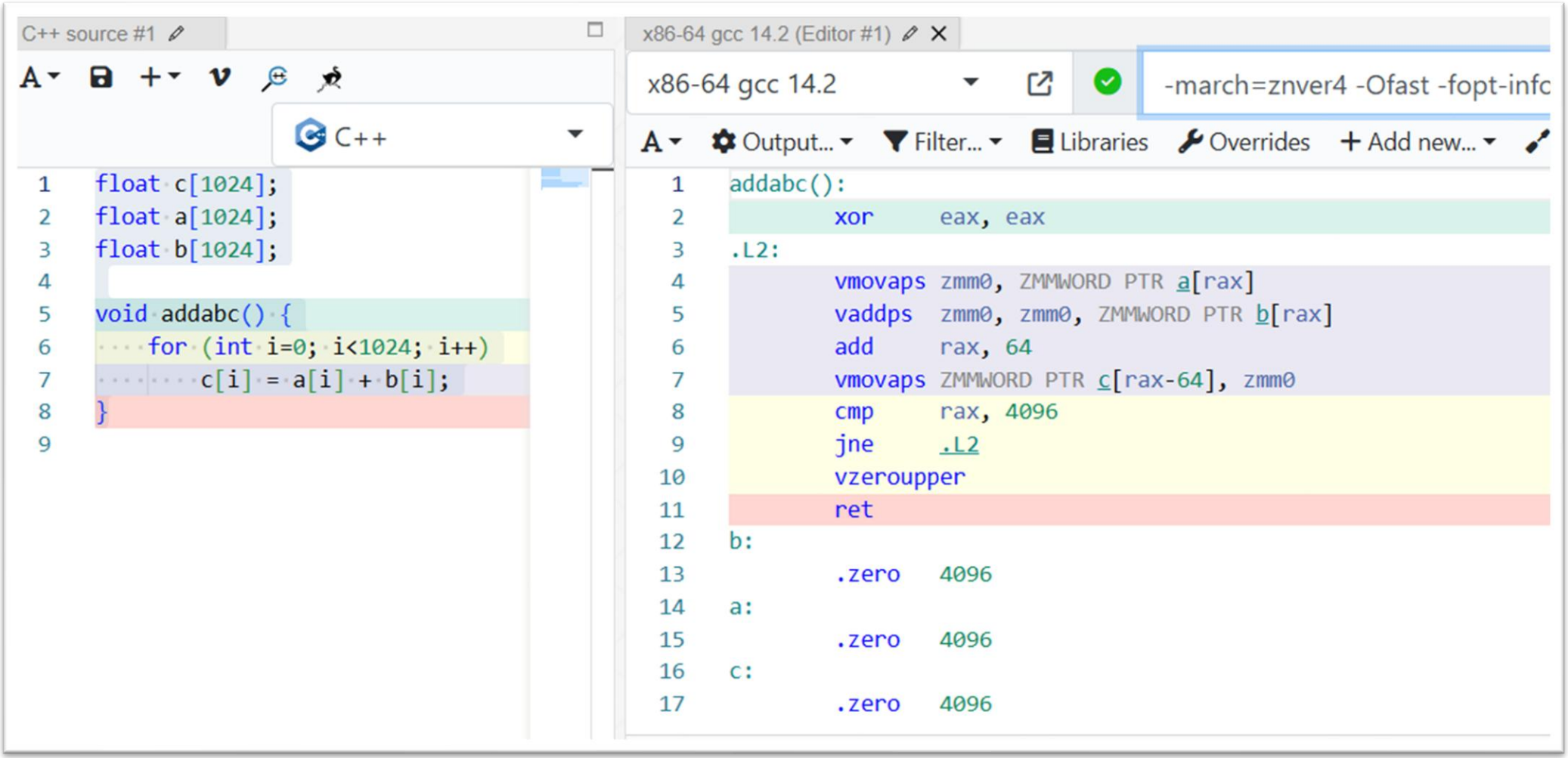
A stretch of Mumbai to Pune expressway near Lonavala.
By neelnimavat - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=49735402



Dubai
https://www.worldatlas.com/articles/countries-with-the-best-roads.html

https://blog.playo.co/cycling-work-insane-idea-boon/

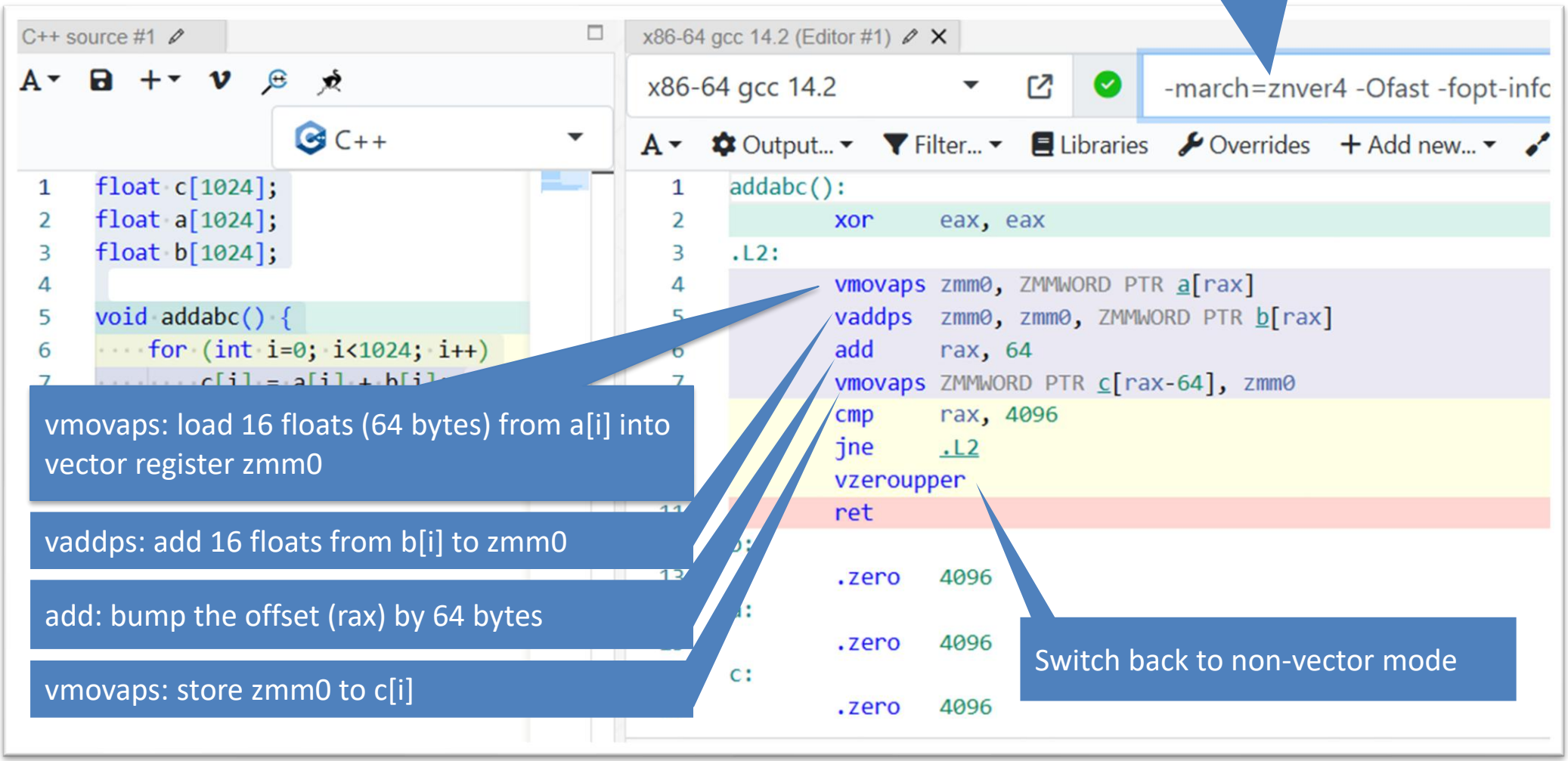
Can we get the compiler to vectorise?



In sufficiently simple cases, no problem:
Gcc reports: **addcba.c:6:20: optimized: loop vectorized using 64 byte vectors**

Can we get the compiler to vectorise?

Tell the compiler to generate code for AMD Zen 4 which has AVX512



vmovaps: load 16 floats (64 bytes) from a[i] into vector register zmm0

vaddps: add 16 floats from b[i] to zmm0

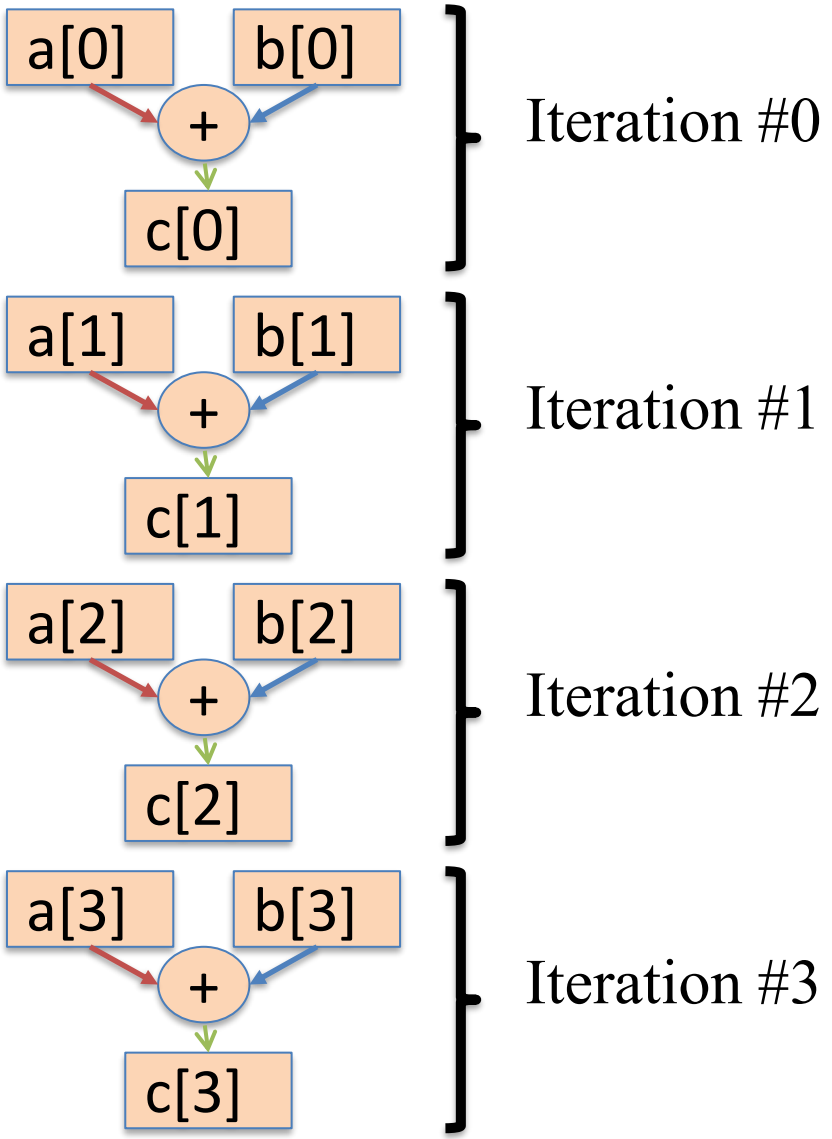
add: bump the offset (rax) by 64 bytes

vmovaps: store zmm0 to c[i]

Switch back to non-vector mode

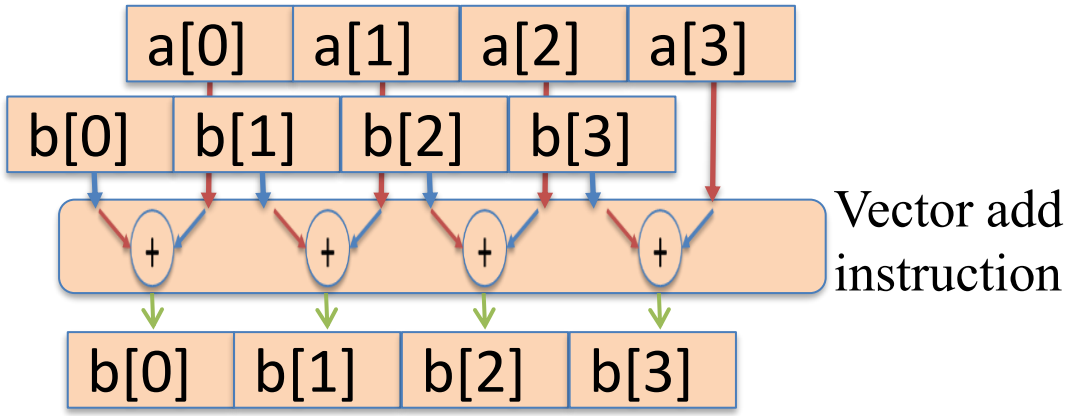
In sufficiently simple cases, no problem:
Gcc reports: **addcba.c:6:20: optimized: loop vectorized using 64 byte vectors**

How do we know a loop is parallel?



- To use vector instructions, we need to verify that different iterations of the loop are truly parallel

- In this case we can easily see that the dependence arrows do not cross iteration boundaries



- So we can use a vector add

- This case was easy:

```
for (int i=0; i<4; i++)
  c[i] = a[i] + b[i];
```

How much does it help?
First: *without* vectorisation

- Source code:

```
for (int i=0; i<size; i++)
        c[i] = a[i] + b[i];
```

- Processor: AMD Ryzen 9 7940HS (“maple10”)
- Compiler command line:

```
gcc -O1 addcba-perf.c
```

- Generated code – not vectorised:

```
.L3:
        movss   (%rsi,%rax), %xmm0
        addss   (%rcx,%rax), %xmm0
        movss   %xmm0, (%rdi,%rax)
        addq    $4, %rax
        cmpq    %rdx, %rax
        jne     .L3
```

- Performance: 4.8 GFLOPS (4.8*10^9 single precision floating-point operations/second)
- Time per loop iteration: 0.21ns (one clock cycle at 4.8GHz, 1 result per iteration)

How much does it help?
This time *with* vectorisation

- Source code:

```
for (int i=0; i<size; i++)
        c[i] = a[i] + b[i];
```

- Processor: AMD Ryzen 9 7940HS (“maple10”)
- Compiler command line:

```
gcc -Ofast -march=znver4 addcba-perf.c
```

- Generated code:

```
.L4:
        vmovaps (%r8,%rax), %zmm1
        vaddps  (%rdi,%rax), %zmm1, %zmm0
        vmovaps %zmm0, (%rsi,%rax)
        addq    $64, %rax
        cmpq    %rax, %rdx
        jne     .L4
```

- Performance: 34.8 GFLOPS (single precision)
- Time per loop iteration: 0.45ns (two clock cycles, 16 results per iteration)

How much does it help?
This time *with* vectorisation

- Source code:

```
for (int i=0; i<size; i++)
        c[i] = a[i] + b[i];
```

- Processor: AMD Ryzen 9 7940HS (“maple10”)
- Compiler command line:

```
gcc -Ofast -march=znver4 addcba-perf.c
```

- Generated code:

```
.L4:
        vmovaps  (%r8,%rax), %zmm1
        vaddps   (%rdi,%rax), %zmm1, %zmm0
        vmovaps %zmm0, (%rsi,%rax)
        addq     $64, %rax
        cmpq     %rax, %rdx
        jne      .L4
```

Speed of light is 30cm/ns.

So this machine completes about three iterations in the time it takes the light to get from your computer screen to your eyes

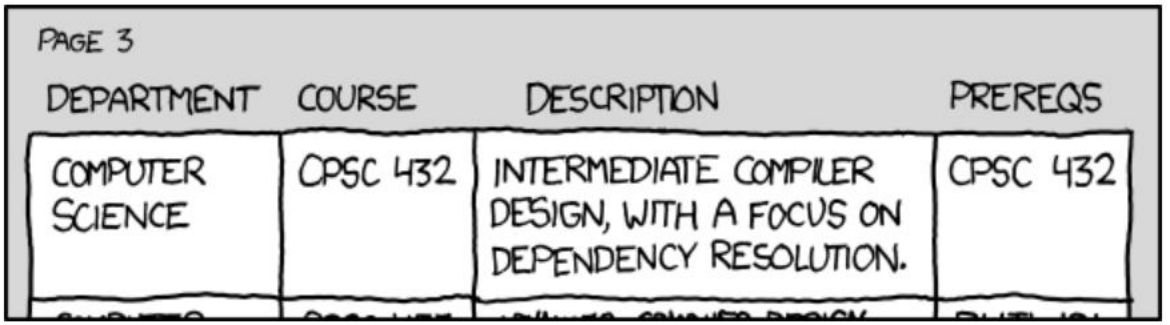
- Performance: 34.8 GFLOPS (single precision)
- Time per loop iteration: 0.45ns (two clock cycles, 16 results per iteration)

Compilers - Chapter 8:

Loop scheduling optimisations

Part 2: Determining whether a loop can be executed in parallel

- Lecturer:
  - Paul Kelly (p.kelly@imperial.ac.uk)



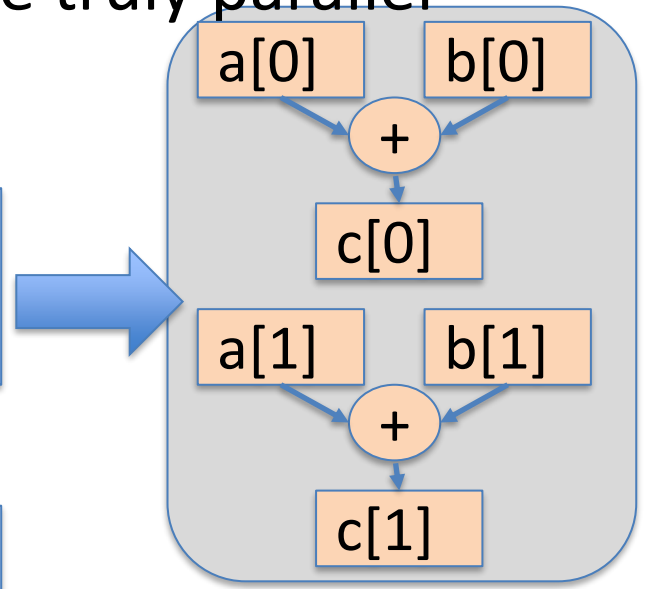
(It turns out that this cartoon is good for almost any compilers topic, definitely this one) https://xkcd.com/754/

But that example was *obviously* parallel?

- To use vector instructions, we need to verify that different iterations of the loop are truly parallel

- This case was easy:

```
for (int i=0; i<1024; i++)        P
  c[i] = a[i] + b[i];
```



- How about this one?

```
for (int i=0; i<1024; i++)        Q
  c[i] = c[i-1] + b[i];
```

But that example was *obviously* parallel?

- To use vector instructions, we need to verify that different iterations of the loop are truly parallel

- This case was easy:

```
for (int i=0; i<1024; i++)        P
  c[i] = a[i] + b[i];
```

- How about this one?

```
for (int i=0; i<1024; i++)        Q
  c[i] = c[i-1] + b[i];
```

- And this?

```
for (int i=0; i<1024; i+=2)       R
  c[i] = c[i-1] + b[i];
```

“Loop-carried dependence”

- Consider this example:

```
for (int i=1; i<8; i++)
  c[i] = c[i-1] + b[i];
```

Q

- What does it *do*?

“Loop-carried dependence”

- Consider this example:

```
for (int i=1; i<8; i++)
  c[i] = c[i-1] + b[i];
```

Q

- When executed we get:

```
c[1] = c[0] + b[1];
c[2] = c[1] + b[2];
c[3] = c[2] + b[3];
c[4] = c[3] + b[4];
c[5] = c[4] + b[5];
c[6] = c[5] + b[6];
c[7] = c[6] + b[7];
```

“Loop-carried dependence”

- Consider this example:

```
for (int i=1; i<8; i++)
  c[i] = c[i-1] + b[i];
```

Q

Each iteration produces a value that is used in the next iteration

- When executed we get:

```
c[1] = c[0] + b[1];
c[2] = c[1] + b[2];
c[3] = c[2] + b[3];
c[4] = c[3] + b[4];
c[5] = c[4] + b[5];
c[6] = c[5] + b[6];
c[7] = c[6] + b[7];
```

The dependence arrows go from one iteration to the next

The dependence is *carried* by the loop

“Loop-carried dependence”

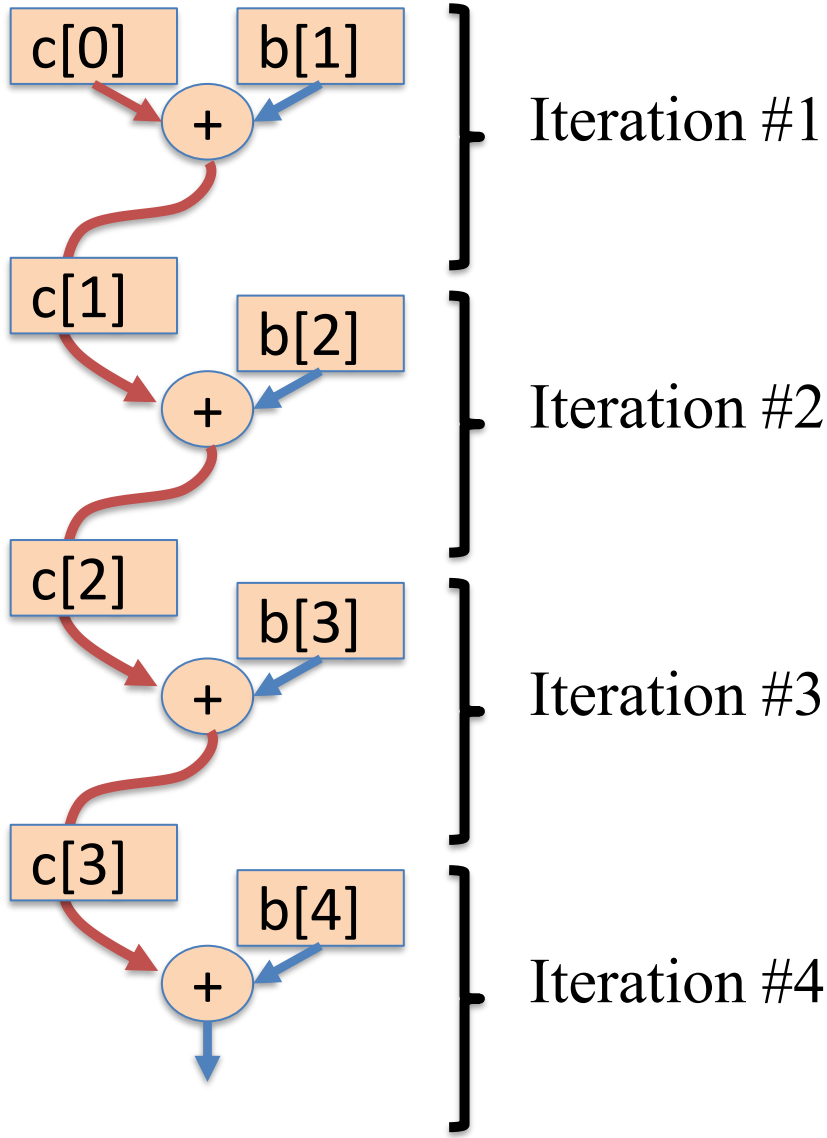
- Consider this example:

```
for (int i=1; i<8; i++)
  c[i] = c[i-1] + b[i];
```

Q

- When executed we get:

```
c[1] = c[0] + b[1];
c[2] = c[1] + b[2];
c[3] = c[2] + b[3];
c[4] = c[3] + b[4];
c[5] = c[4] + b[5];
c[6] = c[5] + b[6];
c[7] = c[6] + b[7];
```



"Loop-carried dependence"

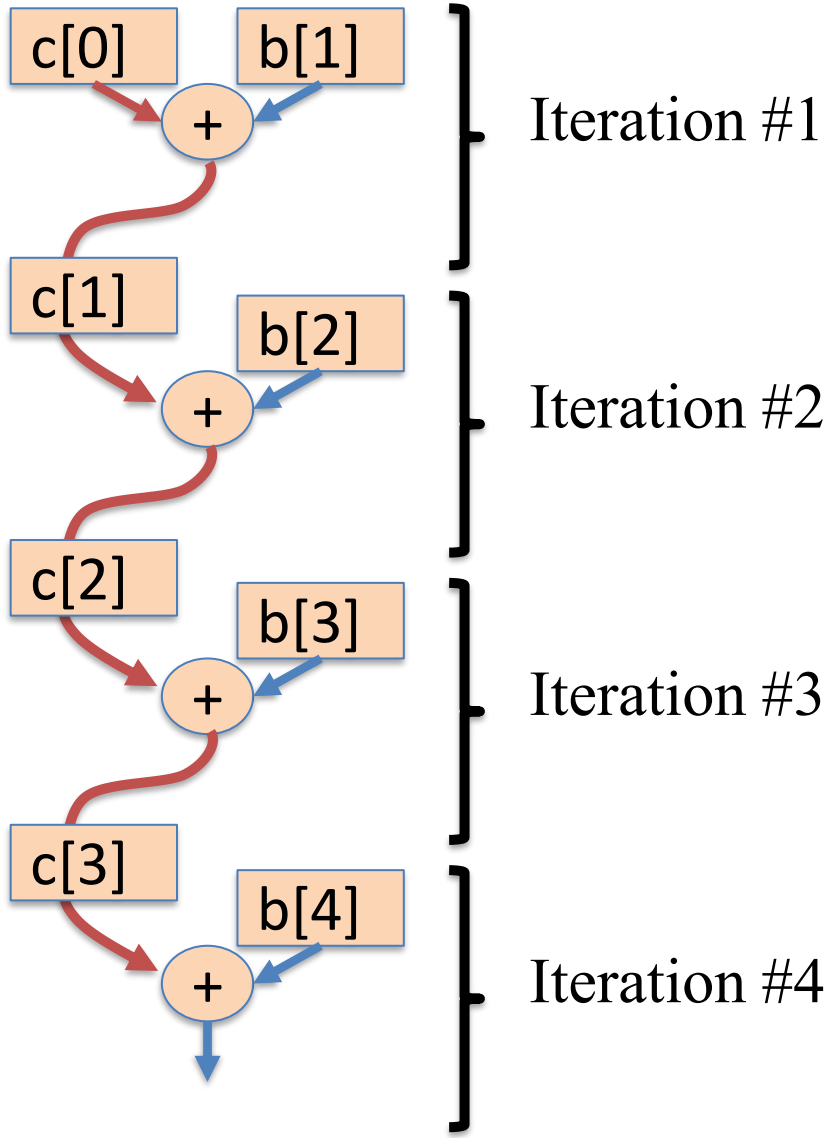
- Consider this example:

```
for (int i=1; i<8; i++)
  c[i] = c[i-1] + b[i];
```

Q

- When executed we get:

```
c[1] = c[0] + b[1];
c[2] = c[1] + b[2];
c[3] = c[2] + b[3];
c[4] = c[3] + b[4];
c[5] = c[4] + b[5];
c[6] = c[5] + b[6];
c[7] = c[6] + b[7];
```



There is a chain of dependence from iteration to iteration

"Loop-carried dependence"

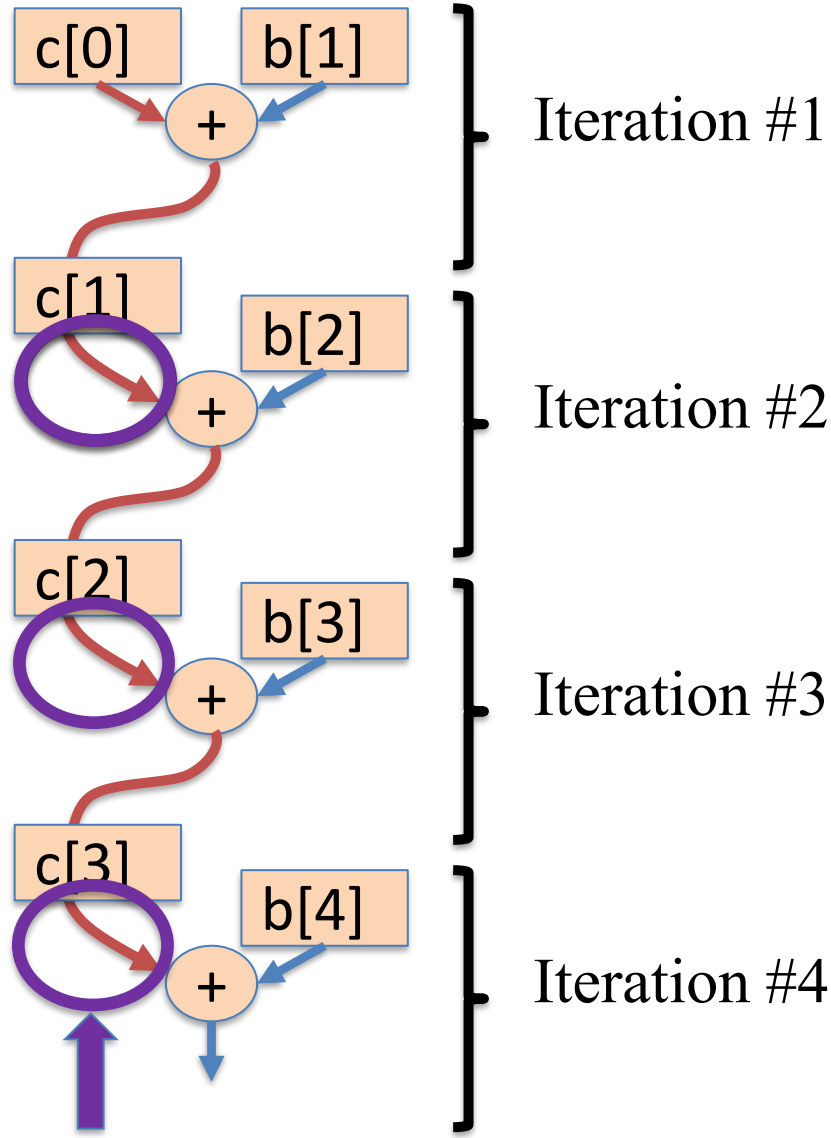
- Consider this example:

```
for (int i=1; i<8; i++)
  c[i] = c[i-1] + b[i];
```

Q

- When executed we get:

```
c[1] = c[0] + b[1];
c[2] = c[1] + b[2];
c[3] = c[2] + b[3];
c[4] = c[3] + b[4];
c[5] = c[4] + b[5];
c[6] = c[5] + b[6];
c[7] = c[6] + b[7];
```



These are the *loop-carried* dependences

So we need a compiler algorithm

- To determine whether there is a loop-carried dependence
- To distinguish, for example, **P**, **Q** and **R**:

```
for (int i=0; i<1024; i++)
  c[i] = a[i] + b[i];
```
P

- No loop-carried dependence
- So iterations can be executed in parallel
- **So vectorisable**

```
for (int i=0; i<1024; i++)
  c[i] = c[i-1] + b[i];
```
Q

- loop-carried dependence
- So iterations cannot be executed in parallel
- **So *not* vectorisable**

```
for (int i=0; i<1024; i+=2)
  c[i] = c[i-1] + b[i];
```
R

- No loop-carried dependence
- So iterations can be executed in parallel
- **So vectorisable**

So we need a compiler algorithm

- To determine whether there is a loop-carried dependence
- To distinguish, for example, **P**, **Q** and **R**:

```
for (int i=0; i<1024; i++)
  c[i] = a[i] + b[i];
```
P

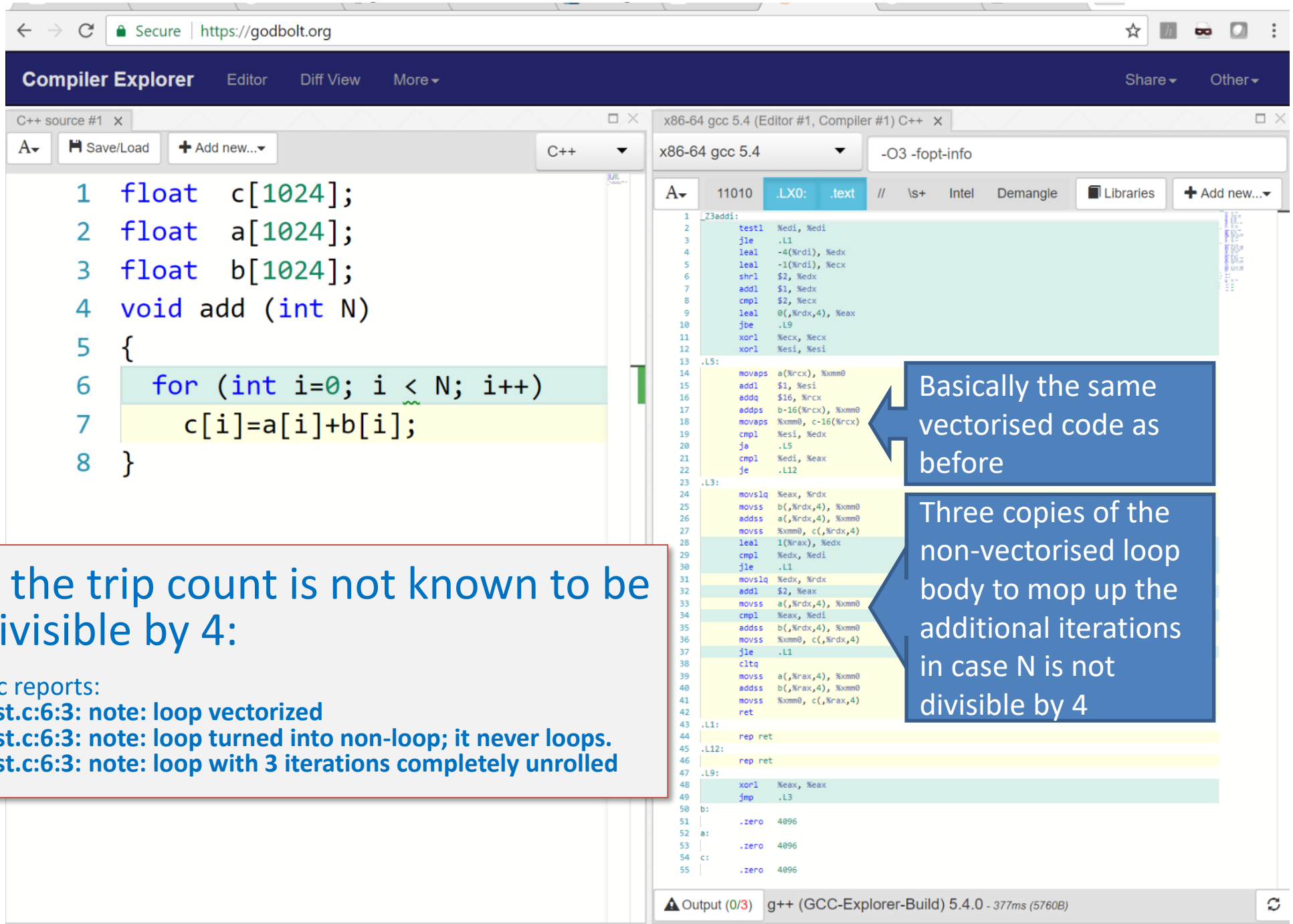
- No loop-carried dependence
- So iterations can be executed in parallel
- **So vectorisable**

```
for (int i=0; i<1024; i++)
  c[i] = c[i-1] + b[i];
```
Q

- loop-carried dependence
- So iterations cannot be executed in parallel
- **So *not* vectorisable**

```
for (int i=0; i<1024; i+=2)
  c[i] = c[i-1] + b[i];
```
R

- No loop-carried dependence
- So iterations can be executed in parallel
- **So vectorisable**
- **(though actually generating efficient vector code for this might be a bit tricky?)**



Basically the same vectorised code as before

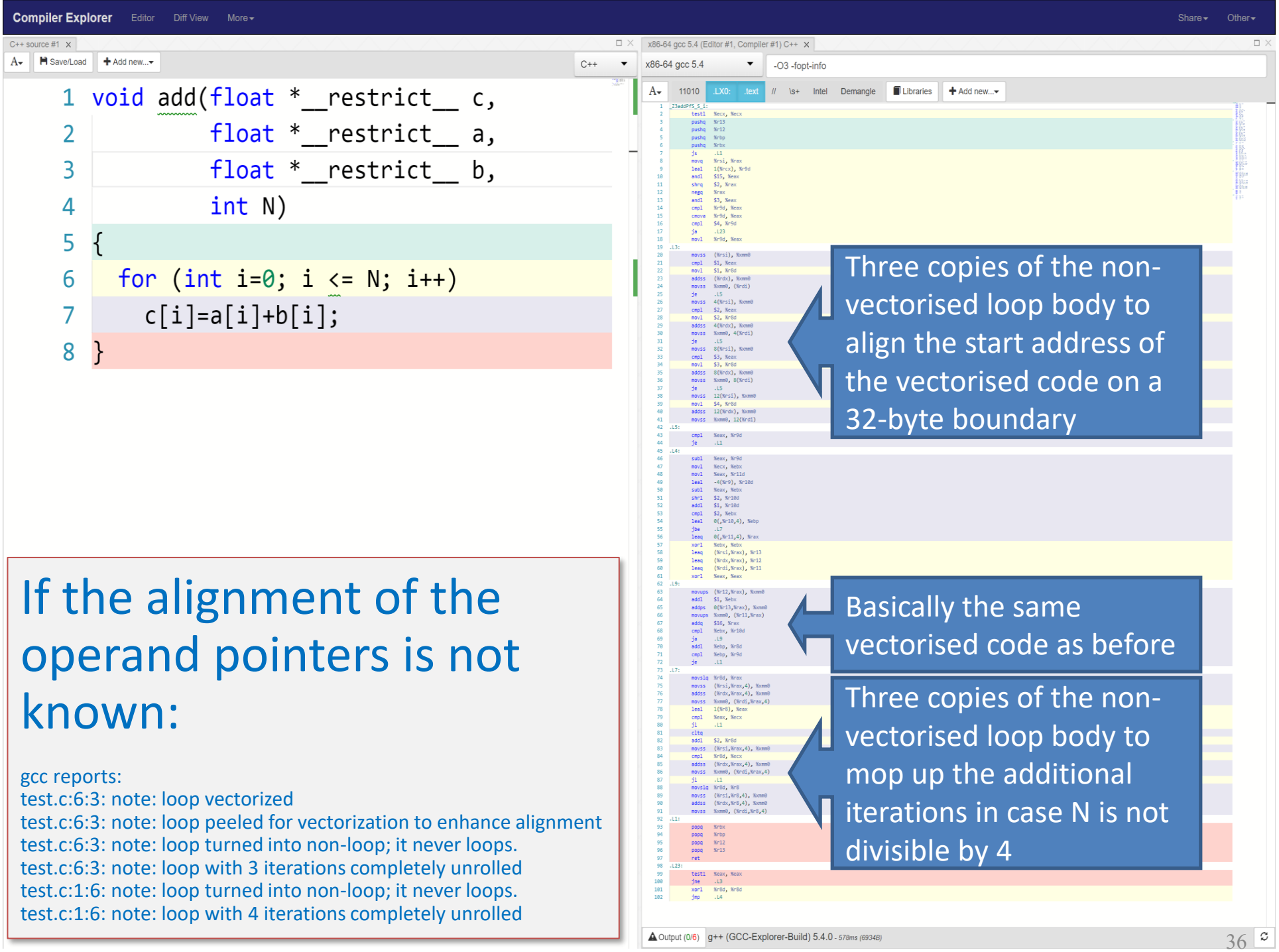
Three copies of the non-vectorised loop body to mop up the additional iterations in case N is not divisible by 4

If the trip count is not known to be divisible by 4:

gcc reports:

test.c:6:3: note: loop vectorized
test.c:6:3: note: loop turned into non-loop; it never loops.
test.c:6:3: note: loop with 3 iterations completely unrolled

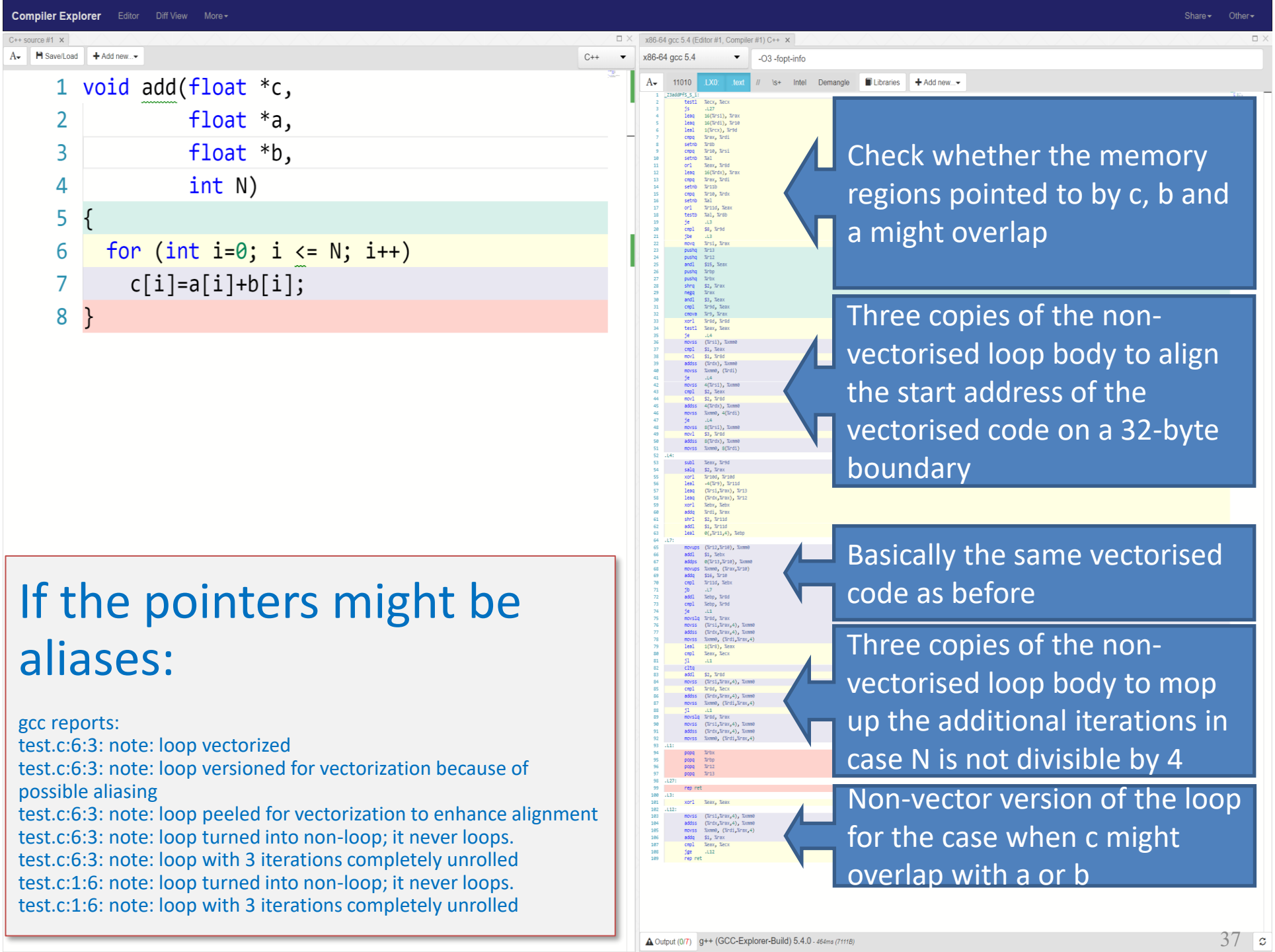
35



If the alignment of the operand pointers is not known:

gcc reports:
test.c:6:3: note: loop vectorized
test.c:6:3: note: loop peeled for vectorization to enhance alignment
test.c:6:3: note: loop turned into non-loop; it never loops.
test.c:6:3: note: loop with 3 iterations completely unrolled
test.c:1:6: note: loop turned into non-loop; it never loops.
test.c:1:6: note: loop with 4 iterations completely unrolled

36

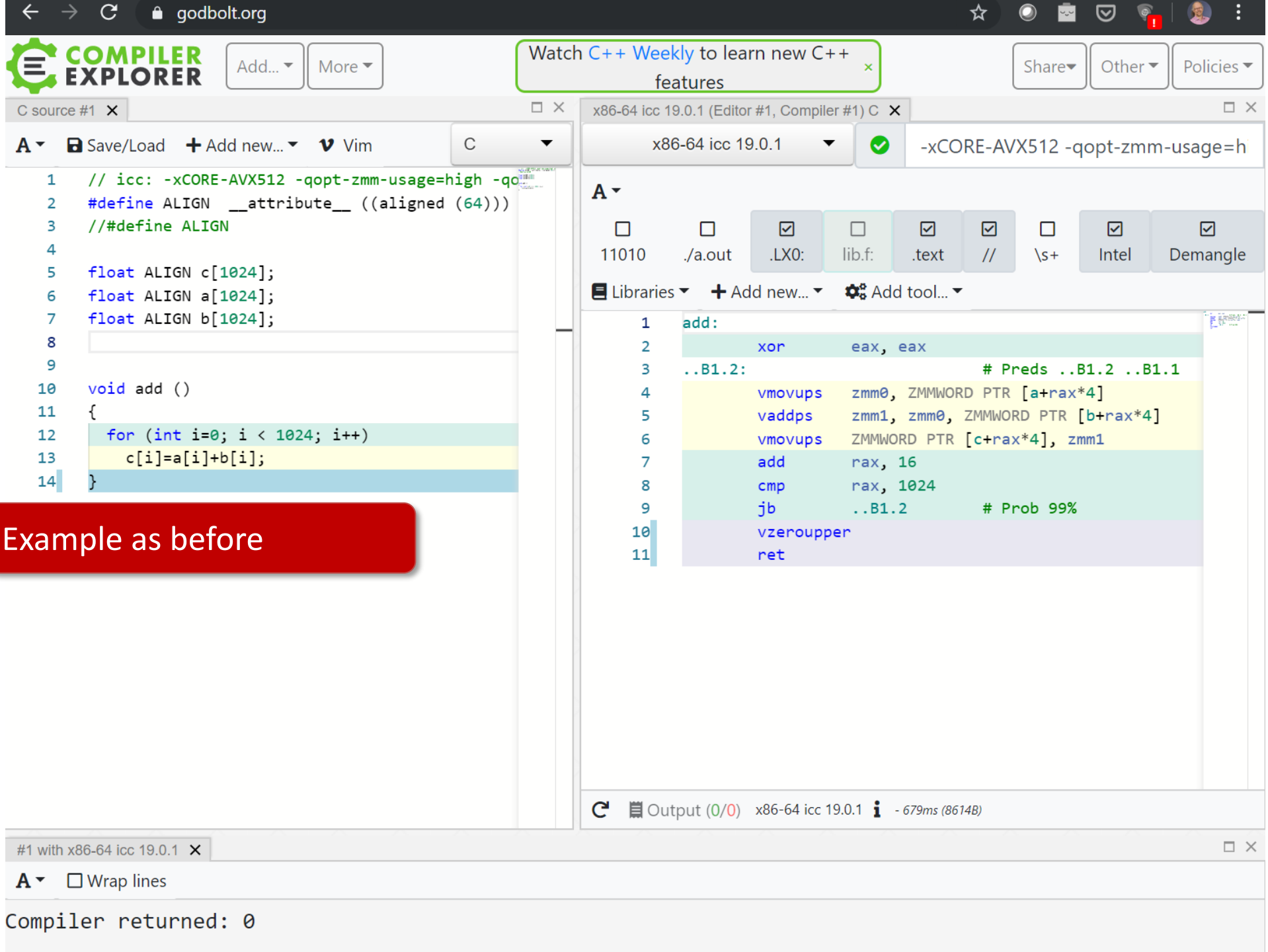

Compiler Explorer
Editor
Diff View
More
Share
Other
C++ source #1
Save/Load
Add new...
C++
1 void add(float *c,
2 float *a,
3 float *b,
4 int N)
5 {
6 for (int i=0; i <= N; i++)
7 c[i]=a[i]+b[i];
8 }
x86-64 gcc 5.4 (Editor #1, Compiler #1) C++
x86-64 gcc 5.4
-O3 -fopt-info
Intel
Demangle
Libraries
Add new...
Check whether the memory regions pointed to by c, b and a might overlap
Three copies of the non-vectorised loop body to align the start address of the vectorised code on a 32-byte boundary
Basically the same vectorised code as before
Three copies of the non-vectorised loop body to mop up the additional iterations in case N is not divisible by 4
Non-vector version of the loop for the case when c might overlap with a or b
If the pointers might be aliases:
gcc reports:
test.c:6:3: note: loop vectorized
test.c:6:3: note: loop versioned for vectorization because of possible aliasing
test.c:6:3: note: loop peeled for vectorization to enhance alignment
test.c:6:3: note: loop turned into non-loop; it never loops.
test.c:6:3: note: loop with 3 iterations completely unrolled
test.c:1:6: note: loop turned into non-loop; it never loops.
test.c:1:6: note: loop with 3 iterations completely unrolled
Output (0/7)
g++ (GCC-Explorer-Build) 5.4.0

37

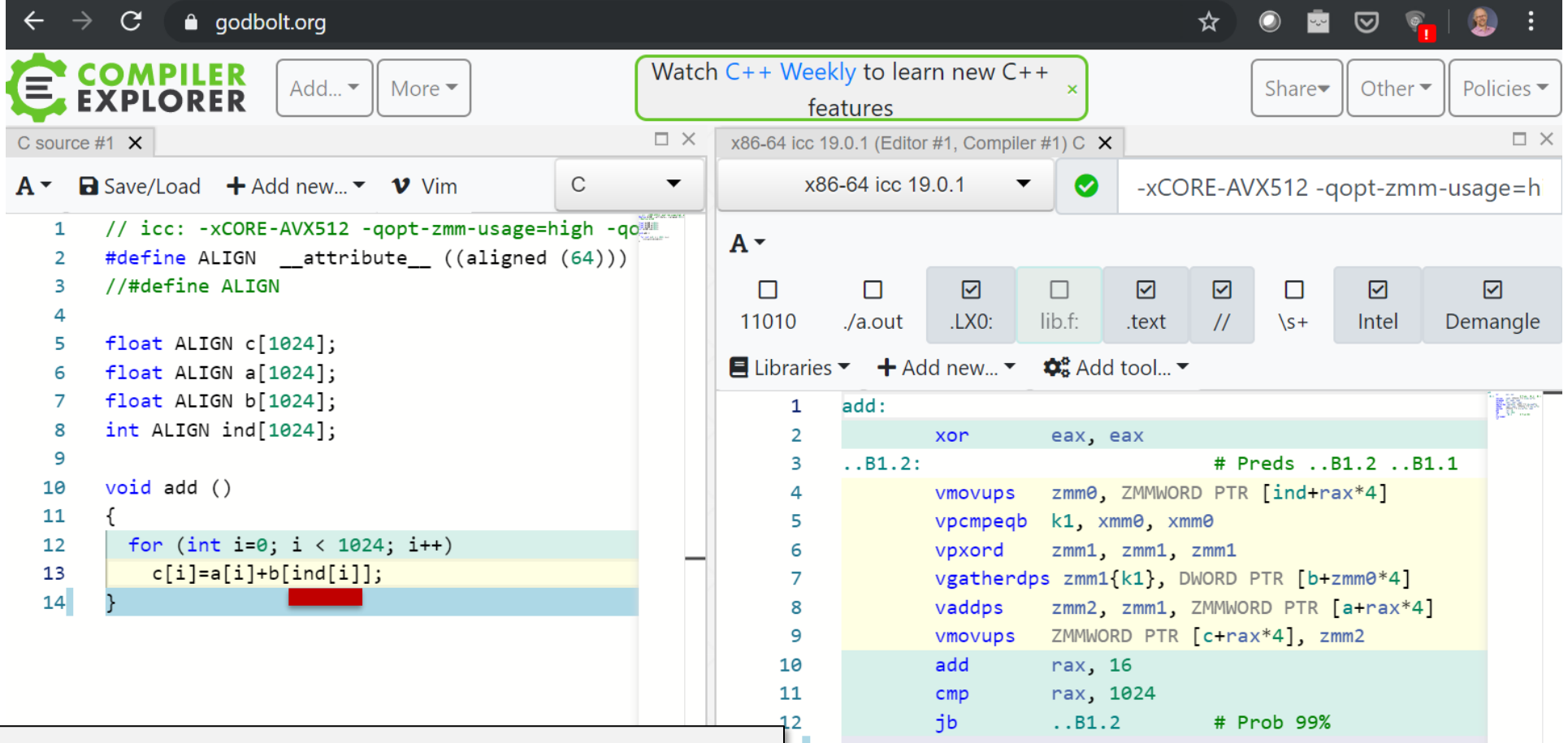
What do we see?

- Actually exploiting vectorisation is a bit tricky even when the dependence analysis is easy

- In the following slides we start with an easily-vectorizable example
- And look at some of the things that make it complicated



Example as before



Indirection: b[ind[]]

We have a register containing a vector of pointers

We need a "gather" instruction:

- A vector load
- That loads from a different address in each lane

(how can this be implemented efficiently??)

40

https://godbolt.org

Startpage Search En... Home - BBC News Staff travel and exp... Shareable Whiteboa... The

COMPILER EXPLORER Add... More Templates

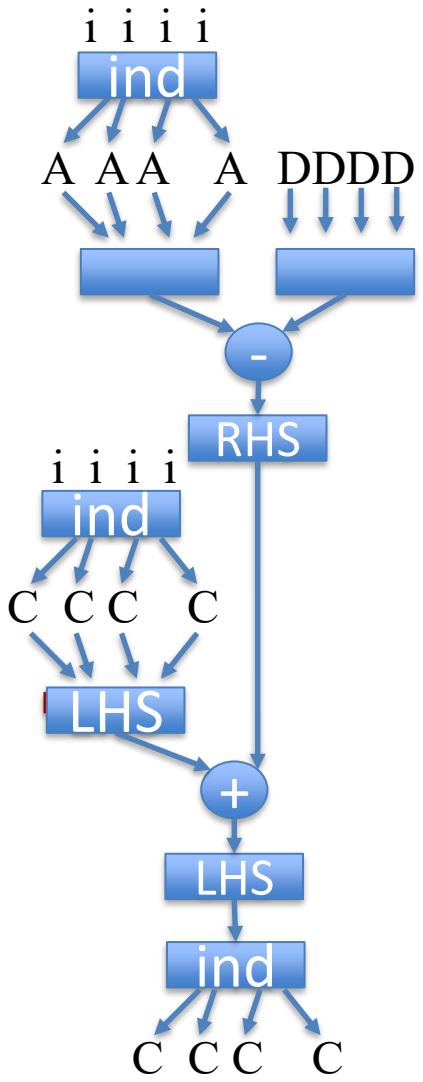
C++ source #1

C++

```
1  // icx -Ofast -march=znver4 -qopt-report
2  #define SIZE 10240
3  #define ALIGN __attribute__ ((aligned(64)))
4
5  int ALIGN A[SIZE];
6  int ALIGN ind[SIZE];
7  int ALIGN C[SIZE];
8  int ALIGN D[SIZE];
9
10 #define IB 32
11 #define JB 32
12
13 void P()
14 {
15   int i, j;
16
17   for (i=0; i<SIZE; i++) {
18     C[ind[i]] += A[ind[i]] - D[i];
19   }
20 }
```

Incrementing through indirection: ind[i]

1. Load a vector ind[i:i+16]
2. Gather a vector A[ind[i:i+16]
3. Subtract the D[i] values:
4. RHS[0:16]=A[ind[i:i+16]] – D[i:i+16]
5. Gather the LHS[0:16] = C[ind[i:i+16]]
6. Add (+=): LHS[0:16] += RHS[0:16]
7. Scatter: C[ind[i:i+16]] = LHS[0:16]



What would happen if there were duplicate indices in ind?

https://godbolt.org

Startpage Search En... | Home - BBC News | Staff travel and exp... | Shareable Whiteboa... | The

COMPILER EXPLORER Add... More Templates

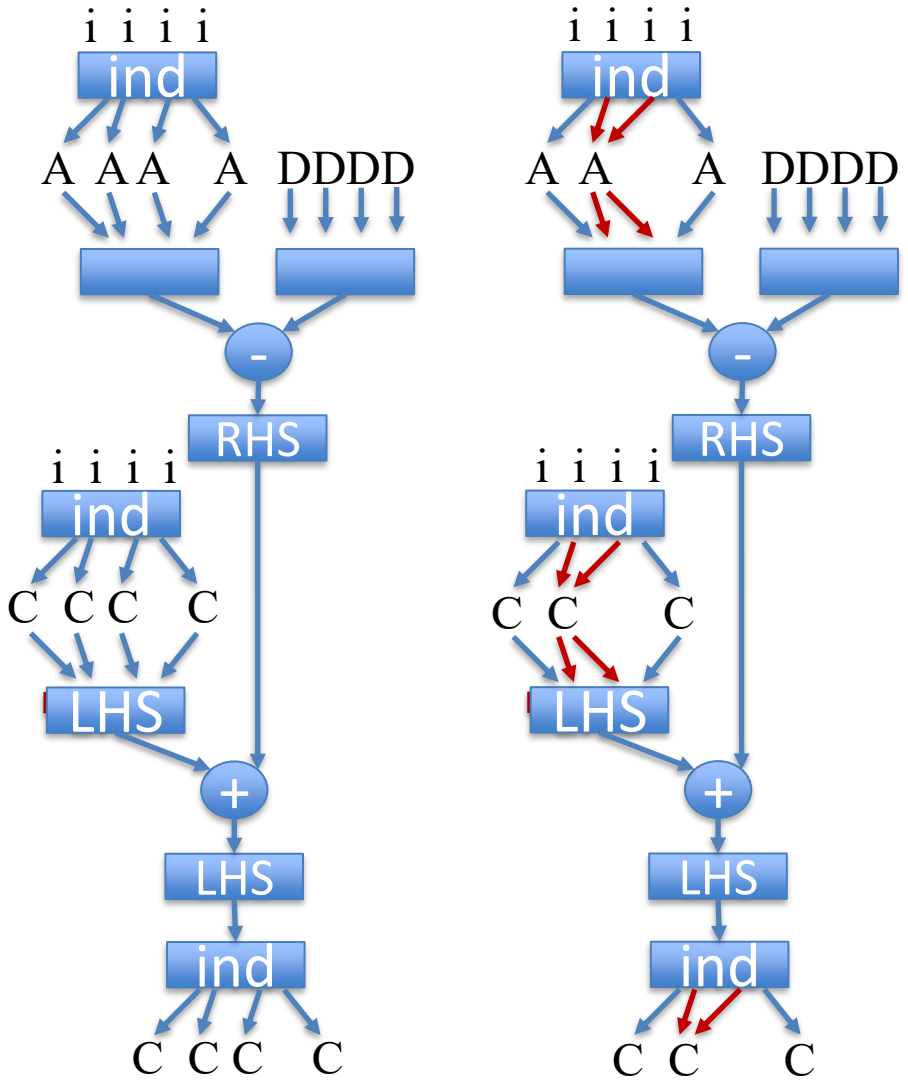
C++ source #1

C++

```
1  // icx -Ofast -march=znver4 -qopt-report
2  #define SIZE 10240
3  #define ALIGN __attribute__ ((aligned(64)))
4
5  int ALIGN A[SIZE];
6  int ALIGN ind[SIZE];
7  int ALIGN C[SIZE];
8  int ALIGN D[SIZE];
9
10 #define IB 32
11 #define JB 32
12
13 void P()
14 {
15   int i, j;
16
17   for (i=0; i<SIZE; i++) {
18     C[ind[i]] += A[ind[i]] - D[i];
19   }
20 }
```

Incrementing through indirection: ind[i]

1. Load a vector ind[i:i+16]
2. Gather a vector A[ind[i:i+16]
3. Subtract the D[i] values:
4. RHS[0:16]=A[ind[i:i+16]] – D[i:i+16]
5. Gather the LHS[0:16] = C[ind[i:i+16]]
6. Add (+=): LHS[0:16] += RHS[0:16]
7. Scatter: C[ind[i:i+16]] = LHS[0:16]



https://godbolt.org

Startpage Search En... | Home - BBC News | Staff travel and exp... | Shareable Whiteboa... | The

COMPILER EXPLORER Add... More Templates

C++ source #1

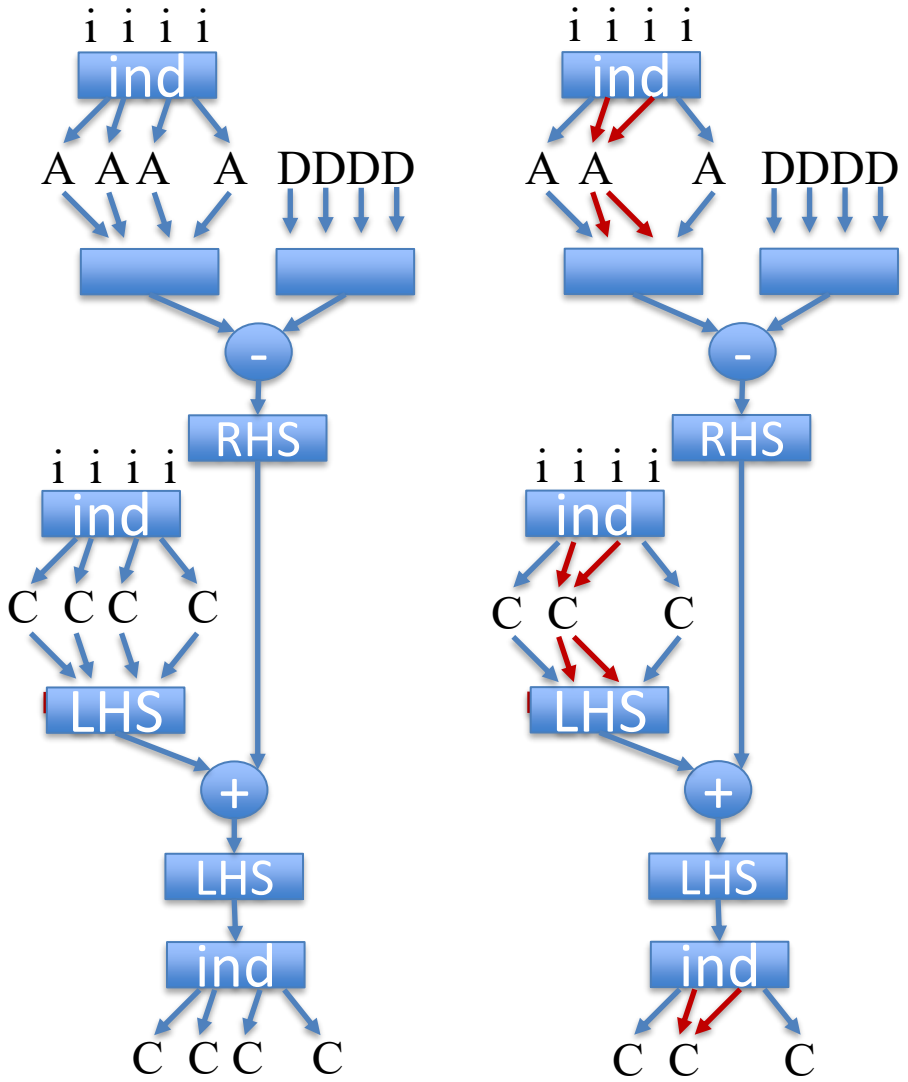
C++

```
1  // icx -Ofast -march=znver4 -qopt-report
2  #define SIZE 10240
3  #define ALIGN __attribute__ ((aligned(64)))
4
5  int ALIGN A[SIZE];
6  int ALIGN ind[SIZE];
7  int ALIGN C[SIZE];
8  int ALIGN D[SIZE];
9
10 #define IB 32
11 #define JB 32
12
13 void P()
14 {
15   int i, j;
16
17   for (i=0; i<SIZE; i++) {
18     C[ind[i]] += A[ind[i]] - D[i];
19   }
20 }
```

Incrementing through indirection: ind[i]

1. Load a vector ind[i:i+16]
2. Gather a vector A[ind[i:i+16]
3. Subtract the D[i] values:
4. RHS[0:16]=A[ind[i:i+16]] – D[i:i+16]
5. Gather the LHS[0:16] = C[ind[i:i+16]]
6. Add (+=): LHS[0:16] += RHS[0:16]
7. Scatter: C[ind[i:i+16]] = LHS[0:16]

What would happen if there were duplicate indices in ind?



It's not parallel! We have to sum two (or more) different values into the same C element

https://godbolt.org

Startpage Search En... | Home - BBC News | Staff travel and exp... | Shareable Whiteboa... | The

COMPILER EXPLORER Add... More Templates

C++ source #1

C++

```
1  // icx -Ofast -march=znver4 -qopt-report
2  #define SIZE 10240
3  #define ALIGN __attribute__ ((aligned(64)))
4
5  int ALIGN A[SIZE];
6  int ALIGN ind[SIZE];
7  int ALIGN C[SIZE];
8  int ALIGN D[SIZE];
9
10 #define IB 32
11 #define JB 32
12
13 void P()
14 {
15   int i, j;
16
17   for (i=0; i<SIZE; i++) {
18     C[ind[i]] += A[ind[i]] - D[i];
19   }
20 }
```

Incrementing through indirection: ind[i]

1. Load a vector ind[i:i+16]
2. Gather a vector A[ind[i:i+16]
3. Subtract the D[i] values:
4. RHS[0:16]=A[ind[i:i+16]] – D[i:i+16]
5. Gather the LHS[0:16] = C[ind[i:i+16]]
6. Add (+=): LHS[0:16] += RHS[0:16]
7. Scatter: C[ind[i:i+16]] = LHS[0:16]

```
.LCPI0_1:
        .long   63
P():
        vpbroadcastd    ymm0, dword ptr [rip + .LCPI0_0]
        vpbroadcastd    ymm1, dword ptr [rip + .LCPI0_1]
        vpcmpeqd        ymm2, ymm2, ymm2
        xor     eax, eax
        jmp     .LBB0_1
.LBB0_7:
        vpcmpeqb        k1, xmm0, xmm0
        vpxor   xmm6, xmm6, xmm6
        vpmovqd ymm3, zmm3
        lea     rcx, [rax + 8]
        add     rax, 16
        vpgatherdd      ymm6 {k1}, ymmword ptr [4*ymm4 + C]
        vpcmpeqb        k1, xmm0, xmm0
        vpaddd  ymm4, ymm6, ymm5
        vpscatterdd     ymmword ptr [4*ymm3 + C] {k1}, ymm4
        cmp     rcx, 10232
        jae     .LBB0_8
.LBB0_1:
        vmovdqu ymm4, ymmword ptr [4*rax + ind]
        vpcmpeqb        k1, xmm0, xmm0
        vpxor   xmm5, xmm5, xmm5
        vpmovsxdq       zmm3, ymm4
        vpgatherdd      ymm5 {k1}, ymmword ptr [4*ymm4 + A]
        vpsubd  ymm5, ymm5, ymmword ptr [4*rax + D]
        vpconflictq     zmm6, zmm3
        vplzcntq        zmm6, zmm6
        vpmovqd ymm6, zmm6
        vpxor   ymm7, ymm6, ymm0
        vptest  ymm7, ymm7
        je      .LBB0_4
        vpcmpneqd       k1, ymm6, ymm0
        vpsubd  ymm6, ymm1, ymm6
.LBB0_3:
        vpermd  ymm7, ymm6, ymm5
        vpermd  ymm6 {k1}, ymm6, ymm6
        vpaddd  ymm5 {k1}, ymm5, ymm7
        vpcmpneqd       k1, ymm6, ymm2
        vptest  ymm6, ymm2
        jae     .LBB0_3
.LBB0_4:
        vpcmpeqb        k1, xmm0, xmm0
        vpxor   xmm6, xmm6, xmm6
        vpmovqd ymm3, zmm3
        vpgatherdd      ymm6 {k1}, ymmword ptr [4*ymm4 + C]
        vpcmpeqb        k1, xmm0, xmm0
        vpaddd  ymm4, ymm6, ymm5
        vpxor   xmm5, xmm5, xmm5
        vpscatterdd     ymmword ptr [4*ymm3 + C] {k1}, ymm4
        vpcmpeqb        k1, xmm0, xmm0
        vmovdqu ymm4, ymmword ptr [4*rax + ind+32]
        vpmovsxdq       zmm3, ymm4
        vpgatherdd      ymm5 {k1}, ymmword ptr [4*ymm4 + A]
        vpsubd  ymm5, ymm5, ymmword ptr [4*rax + D+32]
        vpconflictq     zmm6, zmm3
        vplzcntq        zmm6, zmm6
        vpmovqd ymm6, zmm6
        vpxor   ymm7, ymm6, ymm0
        vptest  ymm7, ymm7
        je      .LBB0_7
        vpcmpneqd       k1, ymm6, ymm0
        vpsubd  ymm6, ymm1, ymm6
.LBB0_6:
        vpermd  ymm7, ymm6, ymm5
        vpermd  ymm6 {k1}, ymm6, ymm6
        vpaddd  ymm5 {k1}, ymm5, ymm7
        vpcmpneqd       k1, ymm6, ymm2
        vptest  ymm6, ymm2
        jae     .LBB0_6
        jmp     .LBB0_7
.LBB0_8:
        vzeroupper
        ret
```

Add RHS into LHS

RHS

Count conflicts

If no conflicts the skip to fast case

Add conflicting lanes' values sequentially

Unrolled copy

Add RHS into LHS
Scatter back

Start next iteration

vpconflictq instruction checks for duplicate values in ind[i:i+16]

Not examinable

If found, we branch to a loop over each distinct value

Roughly…

This is addressed by AVX512 "conflict detect" instructions which enable us to catch duplicates and serialise where needed

Health warning

- Automatic discovery of parallelism has a bad reputation
  - Deservedly! It looks great on simple examples
  - But real code has complexity that means it often just doesn't happen
- But in some application domains it can really work
- And some programming languages make it easier, maybe!
  - Functional languages lack anti- and output-dependences (but tend to add higher-order functions and lazy evaluation)
  - Some languages control pointer ownership and aliasing
  - Some programming models discourage explicit loops and explicit elementwise subscripting

So: we need a compiler algorithm to determine whether a loop is parallel…

Dependence

How?

- Define:
  - IN(S): set of memory locns which might be read by some execn of statement S
  - OUT(S): set of memory locns which might be written by some execn of statement S
- Reordering is constrained by dependences;
- There are four types:
  - Data ("true") dependence: S1 δ S2
    - OUT(S1) ∩ IN(S2)
  - Anti dependence: S1 $\bar{\delta}$ S2
    - IN(S1) ∩ OUT(S2)
  - Output dependence: S1 δ^o S2
    - OUT(S1) ∩ OUT(S2)
  - Control dependence: S1 δ^c S2

("S1 must write something before S2 can read it")

("S1 must read something before S2 overwrites it")

("If S1 and S2 might both write to a location, S2 must write after S1")

("S1 determines whether S2 should execute")

- These are static analogues of the dynamic RAW, WAR, WAW and control hazards which have to be considered in processor architecture

Loop-carried dependences

- **Recall:**

```
S1 :  A[0] := 0
      for I = 1 to 8
S2 :    A[I] := A[I-1] + B[I]
```

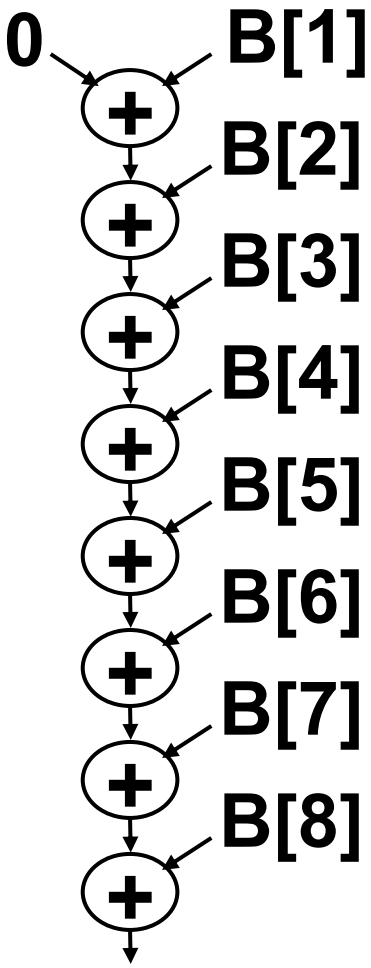
- **What does this loop do?**

B:		**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
A:	**0**								

Loop-carried dependences

- **Recall:**

```
S1 :  A[0] := 0
      for I = 1 to 8
S2 :    A[I] := A[I-1] + B[I]
```



- **What does this loop do?**

B:		1	1	1	1	1	1	1	1
A:	0	1	2						

- **In this case, there is a data dependence**
  - **This is a loop-carried dependence - the dependence spans a loop iteration**
  - **This loop is inherently sequential**

Loop-carried dependences

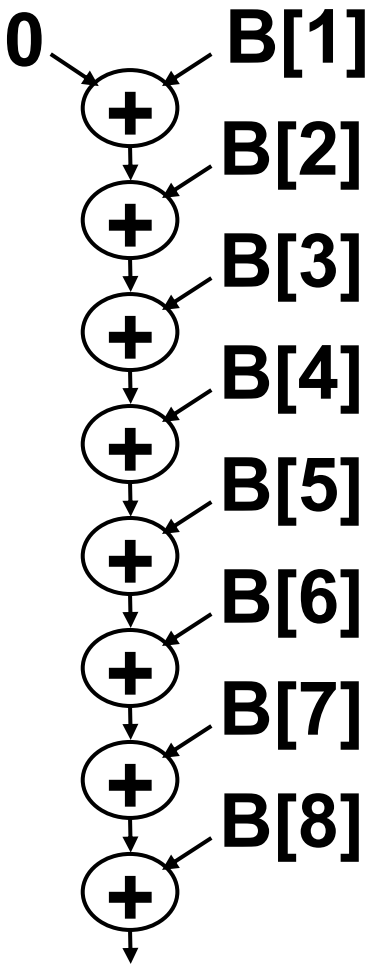
- **Recall:**

```
S1 :   A[0] := 0
       for I = 1 to 8
S2 :     A[I] := A[I-1] + B[I]
```

- **Loop carried:**

$S2^1$: A[1] := A[0] + B[1]
$S2^2$: A[2] := A[1] + B[2]
$S2^3$: A[3] := A[2] + B[3]
$S2^4$: A[4] := A[3] + B[4]
$S2^5$: A[5] := A[4] + B[5]
$S2^6$: A[6] := A[5] + B[6]
$S2^7$: A[7] := A[6] + B[7]
$S2^8$: A[8] := A[7] + B[8]

Dependences cross, from one iteration to next



What is a loop-carried dependence?

- Consider two iterations I^1 and I^2
- A dependence occurs between two statements S_p and S_q (not necessarily distinct), when an assignment in S_p^{I1} refers to the same location as a use in S_q^{I2}

In the example,

```
S1 : A[0] := 0
     for I = 1 to 5
S2 :   A[I] := A[I-1] + B[I]
```

- The assignment is "A[I^1] := ..."
- The use is "... := A[I^2-1] ..."
- These refer to the same location when $I^1 = I^2-1$
- Thus $I^1 < I^2$, ie the assignment is in an earlier iteration

Notation: S_2 $\delta_<$ S_2

Definition: The dependence equation

- **A dependence occurs**
  - **between two statements S_p and S_q (not necessarily distinct),**
  - **when there exists a pair of loop iterations I^1 and I^2,**
  - **such that a memory reference in S_p in I^1 may refer to the same location as a memory reference in S_q in I^2.**

- **This might occur if S_p and S_q refer to some common array A**
- **Suppose S_p refers to $A[\varphi_p(I)]$**
- **Suppose S_q refers to $A[\varphi_q(I)]$**

($\varphi_p(I)$ is some subscript expression involving I)

- **A dependence of some kind occurs between S_p and S_q if there exists a solution to the equation**

$$\varphi_p(I^1) = \varphi_q(I^2)$$

- **for integer values of I^1 and I^2 lying within the loop bounds**

Types of dependence

- If a solution to the dependence equation exists, a dependence of some kind occurs
- The dependence type depends on what solutions exist
- The solutions consist of a set of pairs (I^1, I^2)
- We would appear to have a *data* dependence if

  $A[\phi_p(I)] \in OUT(S_p)$

  and

  $A[\phi_q(I)] \in IN(S_q)$

- But we only really have a data dependence if the assignments *precede* the uses, ie
  - $S_p \ \delta_< \ S_q$
  - if, for each solution pair (I^1, I^2), $I^1 < I^2$

Dependence versus anti-dependence

- If the *uses* precede the *assignments*, we actually have an *anti*-dependence, ie

$$S_p \ \overline{\delta}_< \ S_q$$

if, for each solution pair $(\mathbf{I}^1, \mathbf{I}^2)$, $\mathbf{I}^1 > \mathbf{I}^2$

- In this case we do have a constraint on execution order
- Because we (may) have to read a value before it (may) be overwritten
- And this anti-dependence is loop-carried
- Anti-dependences prevent re-ordering, and multi-thread parallelism

Dependence versus anti-dependence

- If there are some solution pairs (I^1,I^2) with $I^1 < I^2$ and some with $I^1 > I^2$, we write

$$S_p \ \delta_* \ S_q$$

  This represents that we know we must respect execution ordering, even though the compiler is unable to classify the dependence fully

- If, for all solution pairs (I^1,I^2), $I^1 = I^2$, there are dependences *within* an iteration of the loop, but there are no loop-carried dependences:

$$S_p \ \delta_= \ S_q$$

Dependence distance

In many common examples, the set of solution pairs is characterised easily:

- **Definition:** dependence distance
  - **If, for all solution pairs (I^1, I^2),**

    $I^1 = I^2 - k$

    then the dependence distance is k
- **For example in the loop we considered earlier,**

S_1 : `A[0] := 0`

`for I = 1 to 5`

S_2 : `A[I] := A[I-1] + B[I]`

We find that S_2 $\delta_<$ S_2 with dependence distance 1.

- ***((of course there are many cases where the difference is not constant and so the dependence cannot be summarised this way)).***

Reuse distance

- When optimising for cache performance, it is sometimes useful to consider the re-use relationship,
  - $IN(S_1) \cap IN(S_2)$
- Here there is no dependence - it doesn't matter which read occurs first
- Nonetheless, cache performance can be improved by minimising the *reuse distance*
- The reuse distance is calculated essentially the same way
- Eg

```
    for I = 5 to 100
S1:  B[I] := A[I] * 2
S2:  C[I] := A[I-5] * 10
```

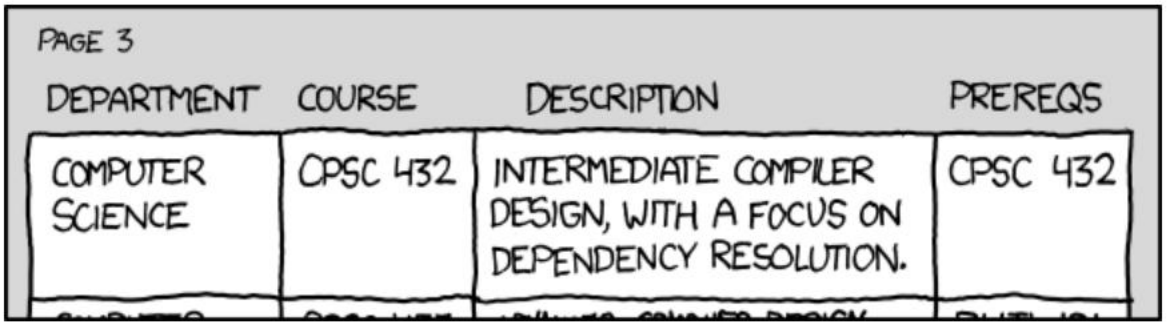
- Here we have a loop-carried reuse with distance 5

Compilers - Chapter 8:

Loop scheduling optimisations

Part 3: Dependence analysis in nested loops

- Lecturer:
  - Paul Kelly (p.kelly@imperial.ac.uk)



(It turns out that this cartoon is good for almost any compilers topic, definitely this one) https://xkcd.com/754/

Nested loops

- Up to now we have looked at single loops
- Now let's generalise to loop "nests"
- We begin by considering a very common dependence pattern, called the "wavefront":

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S:    A[I1,I2] := A[I1 - 1,I2] + A[I1,I2 - 1]
```

- Dependence structure?

Nested loops

- Up to now we have looked at single loops
- Now let's generalise to loop "nests"
- We begin by considering a very common dependence pattern, called the "wavefront":

```
for I = 0 to 3 do
  for J = 0 to 3 do
S:  A[I,J] = A[I-1,J] + A[I,J-1]
```

- Dependence structure?

I is I_1
J is I_2

System of dependence equations

- Consider the dependence equations for this loop nest:

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S:    A[I1,I2] := A[I1 − 1,I2] + A[I1,I2 − 1]
```

- There are two potential dependences arising from the three references to A, so two systems of dependence equations to solve:

1. Between $\texttt{A}[I_1^1, I_2^1]$ and $\texttt{A}[I_1^2 - 1, I_2^2]$:

$$\begin{cases} I_1^1 = I_1^2 - 1 \\ I_2^1 = I_2^2 \end{cases}$$

2. Between $\texttt{A}[I_1^1, I_2^1]$ and $\texttt{A}[I_1^2, I_2^2 - 1]$:

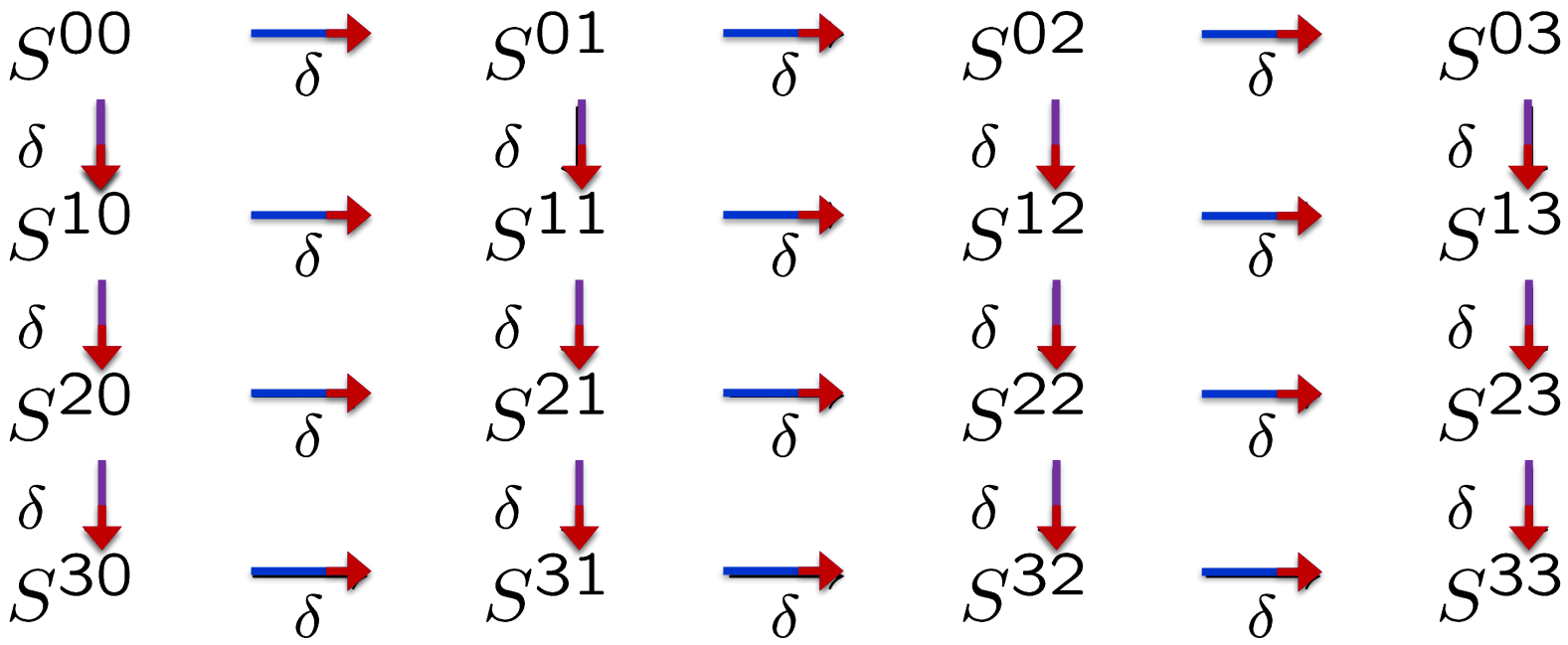
$$\begin{cases} I_1^1 = I_1^2 \\ I_2^1 = I_2^2 - 1 \end{cases}$$

Iteration space graph

- **The same loop:**

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S:   A[I1,I2] := A[I1 − 1,I2] + A[I1,I2 − 1]
```

- **For humans the easy way to understand this loop nest is to draw the *iteration space graph* showing the iteration-to-iteration dependences:**



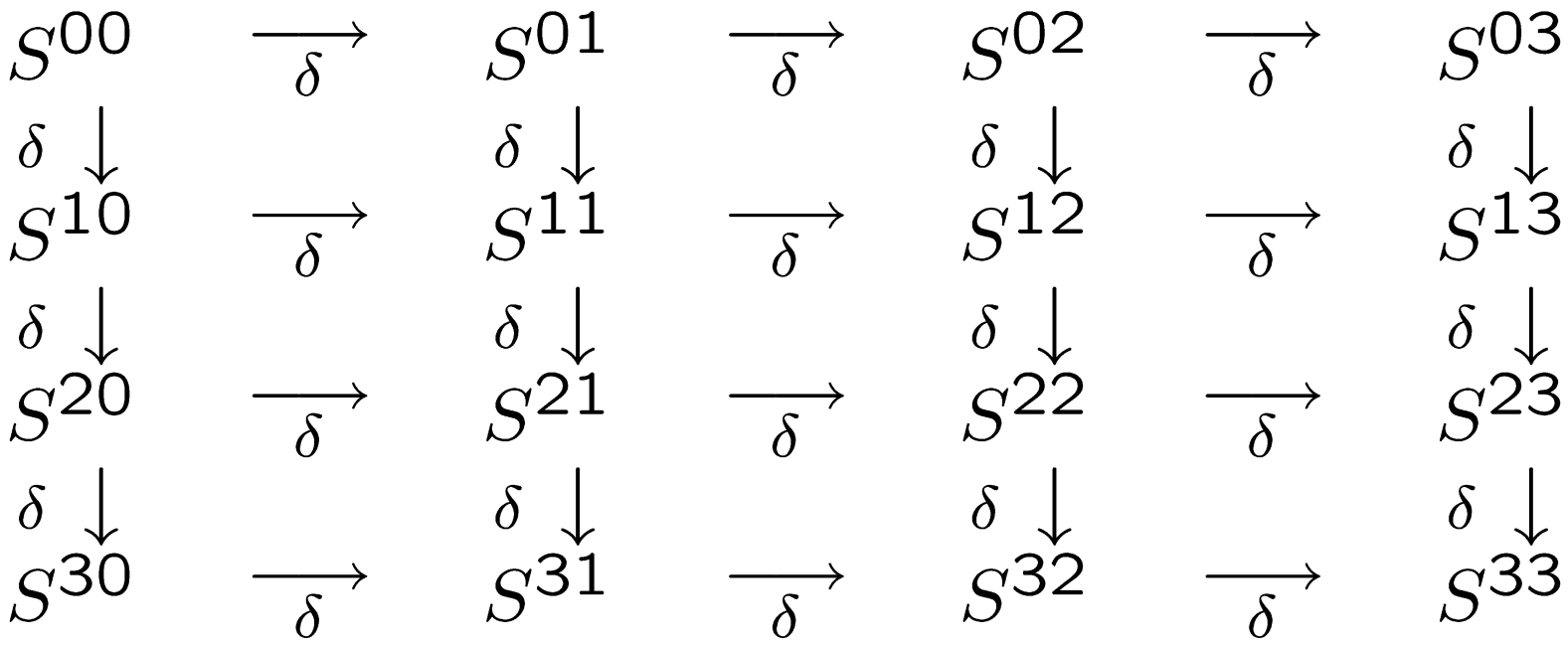
- **The diagram shows an arrow for each solution of each dependence equation.**

Iteration space graph

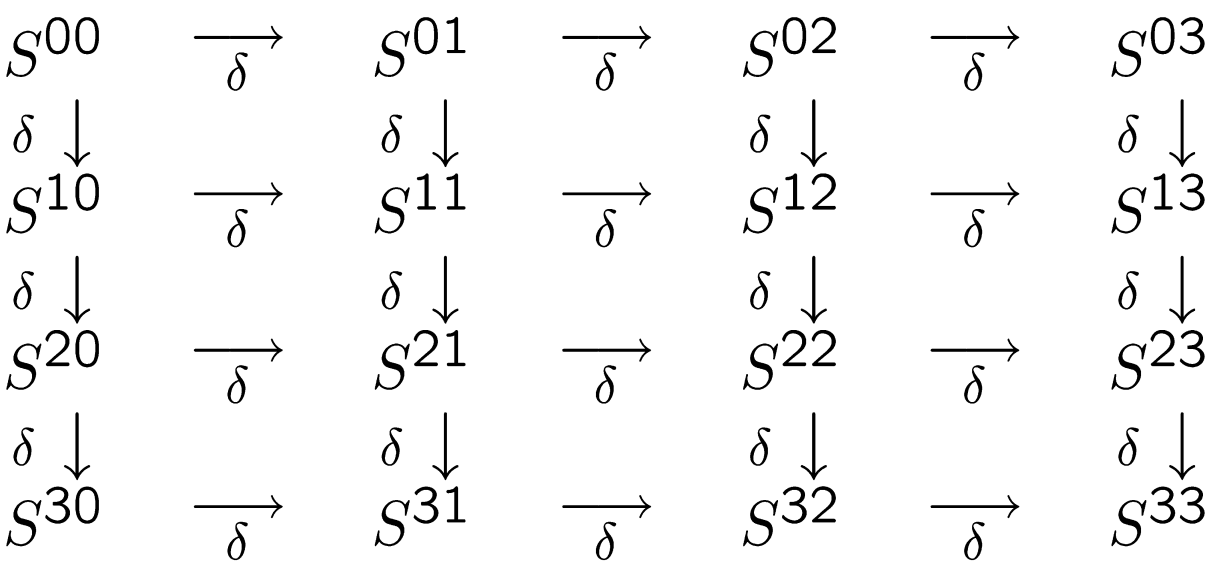
- **The same loop:**

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S:   A[I1,I2] := A[I1 − 1,I2] + A[I1,I2 − 1]
```

- **For humans the easy way to understand this loop nest is to draw the *iteration space graph* showing the iteration-to-iteration dependences:**

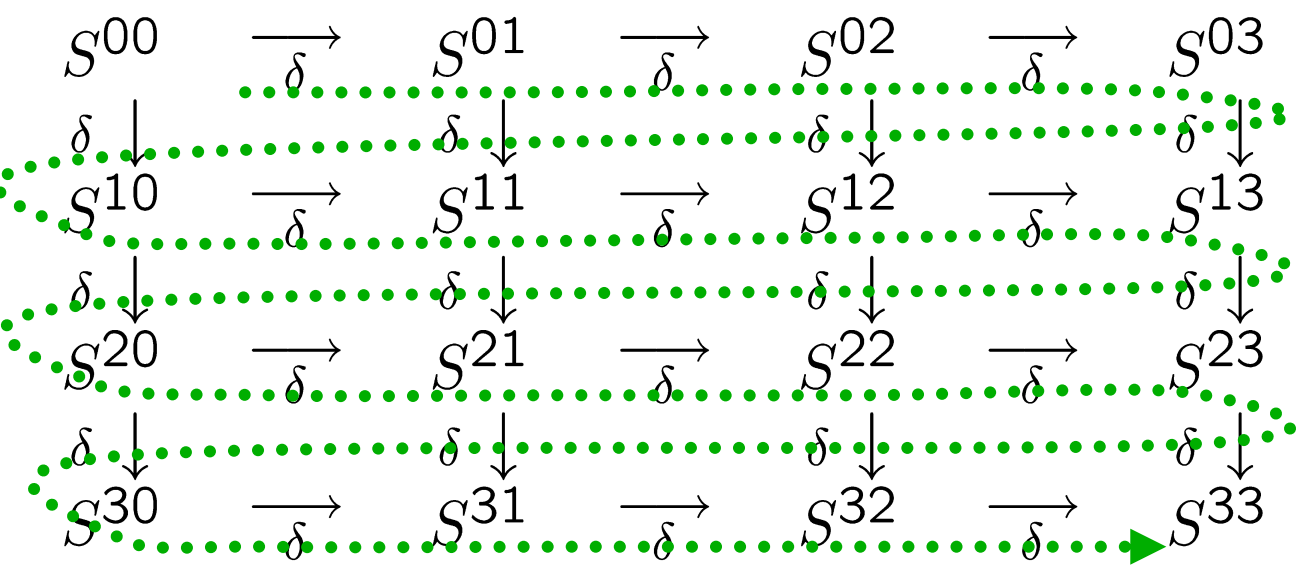


- **The diagram shows an arrow for each solution of each dependence equation. Is there any parallelism?**



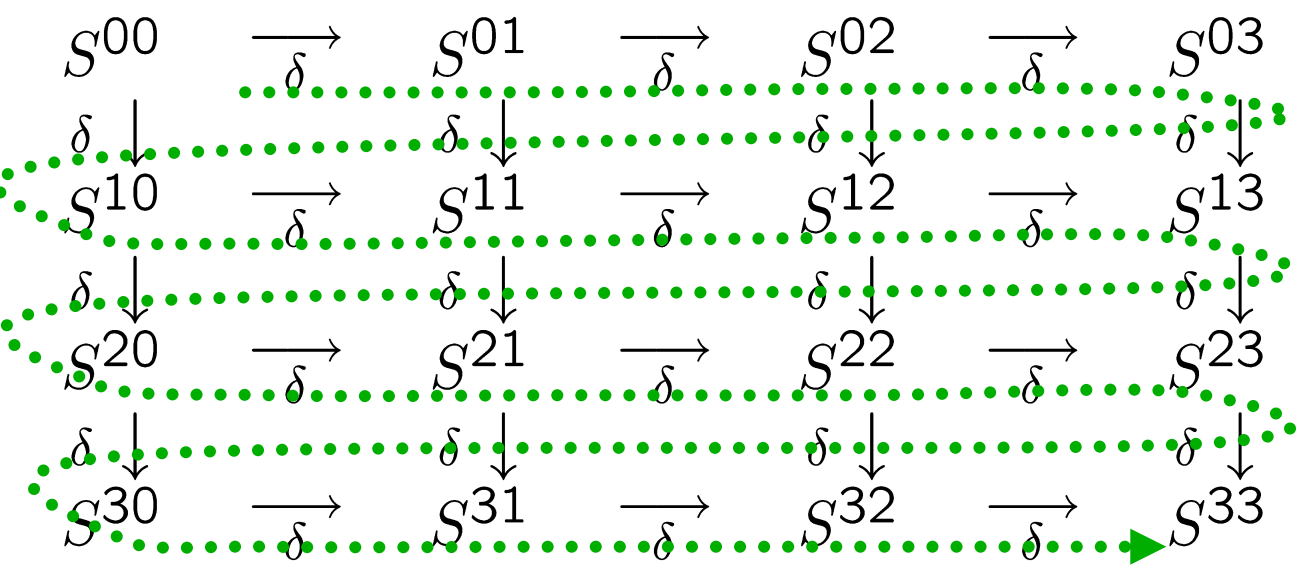

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S:  A[I1,I2] := A[I1 - 1,I2] + A[I1,I2 - 1]
```

- **The inner loop is not vectorisable since there is a dependence chain linking successive iterations.**
  - **(to use a vector instruction, need to be able to operate on each element of the vector in parallel)**
- **Similarly, the outer loop is not parallel**



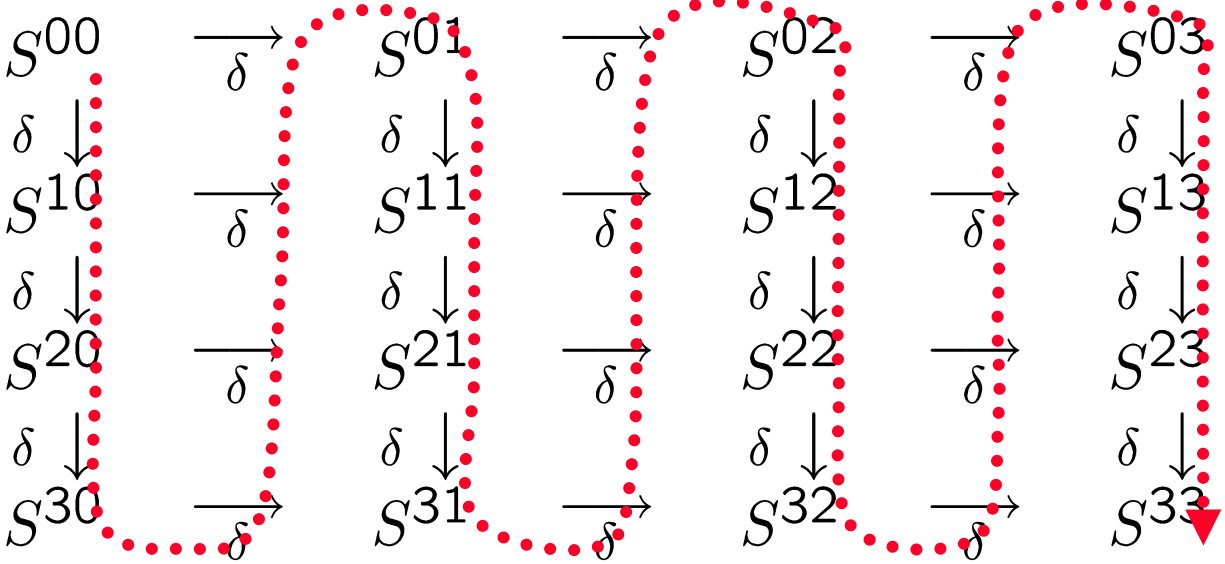

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S:   A[I1,I2] := A[I1 - 1,I2] + A[I1,I2 - 1]
```

- **The inner loop is not vectorisable since there is a dependence chain linking successive iterations.**
  - **(to use a vector instruction, need to be able to operate on each element of the vector in parallel)**
- **Similarly, the outer loop is not parallel**




```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S:   A[I1,I2] := A[I1 - 1,I2] + A[I1,I2 - 1]
```

- **The inner loop is not vectorisable since there is a dependence chain linking successive iterations.**
  - **(to use a vector instruction, need to be able to operate on each element of the vector in parallel)**
- **Similarly, the outer loop is not parallel**
- **This loop nest has two dependence distance vectors:**
  - **(1,0) carried by the outer loop** ↓ **Direction vector: (<,=)**
  - **(0,1) carried by the inner loop** → **Direction vector: (=,<)**




```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S:   A[I1,I2] := A[I1 - 1,I2] + A[I1,I2 - 1]
```

- **The inner loop is not vectorisable since there is a dependence chain linking successive iterations.**
  - **(to use a vector instruction, need to be able to operate on each element of the vector in parallel)**
- **Similarly, the outer loop is not parallel**
- **This loop is *interchangeable*: the top-to-bottom, left-to-right execution order is also valid since all dependence constraints (as shown by the arrows) are still satisfied.**
- **Interchanging the loop does not improve vectorisability or parallelisability**

Interchange: counter-example

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
    A[I1,I2] := A[I1 ± ?, I2 ± ?] + B[I1, I2]
```

Can you think of a loop like this that *cannot* safely be interchanged?

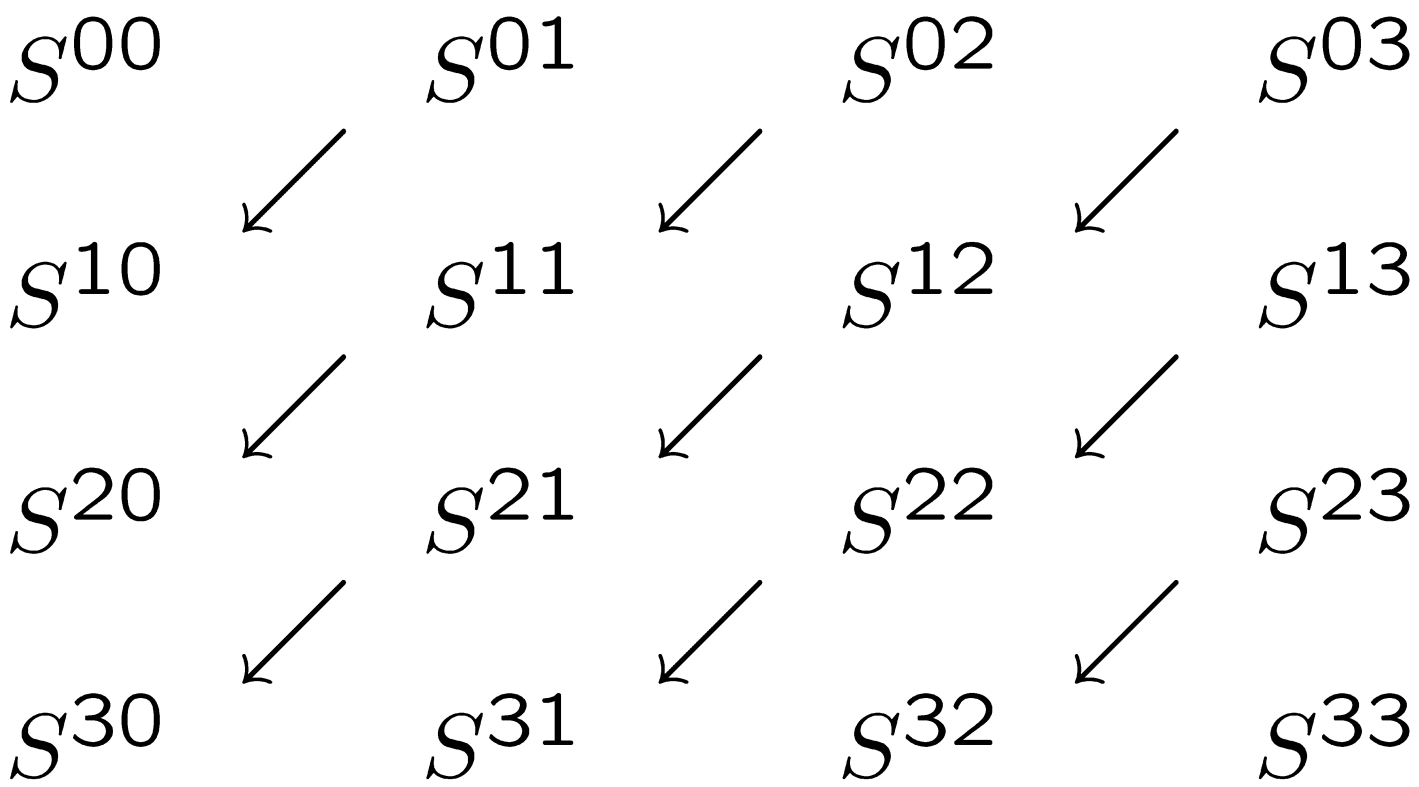
Interchange: counter-example

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
    A[I1,I2] := A[I1 ± 1, I2 ± 1] + B[I1, I2]
```

Can you think of a loop like this that *cannot* safely be interchanged?

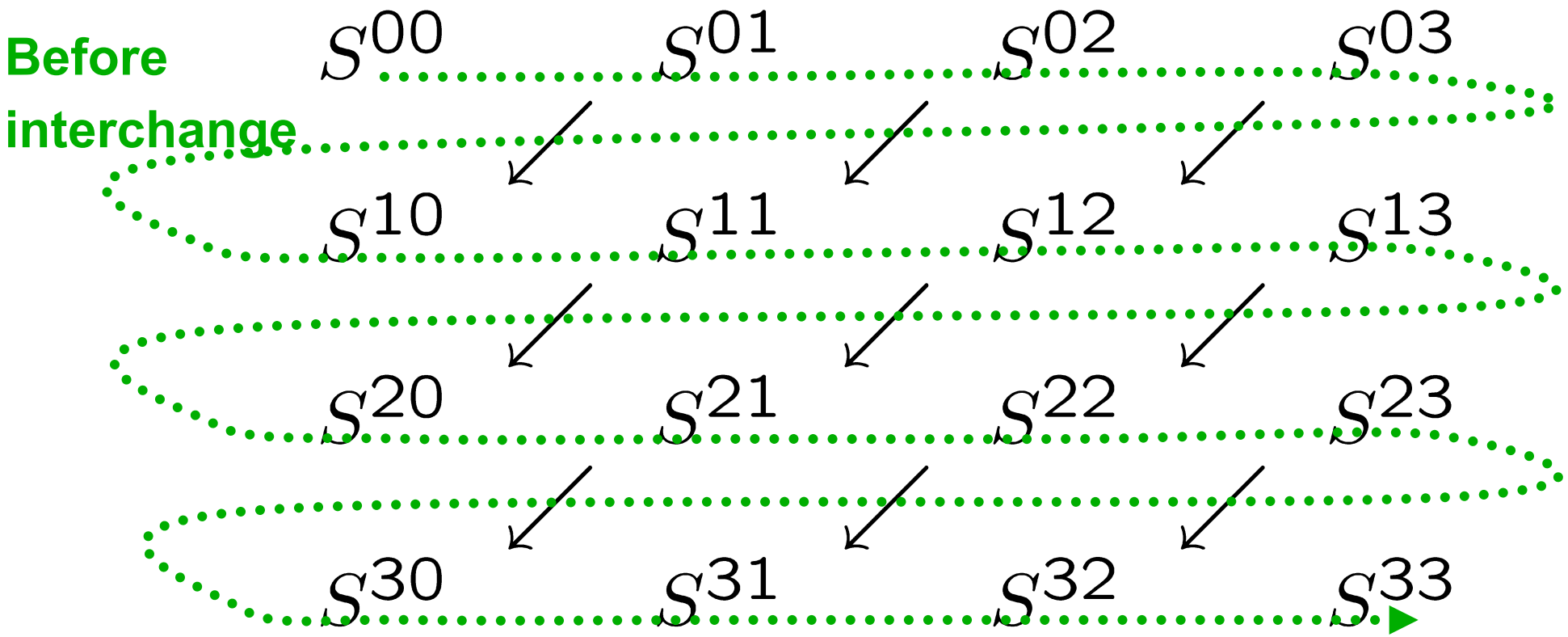
Interchange: counter-example

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
    A[I1,I2] := A[I1 - 1, I2 + 1] + B[I1, I2]
```



Interchange: counter-example

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
    A[I1,I2] := A[I1 - 1, I2 + 1] + B[I1, I2]
```

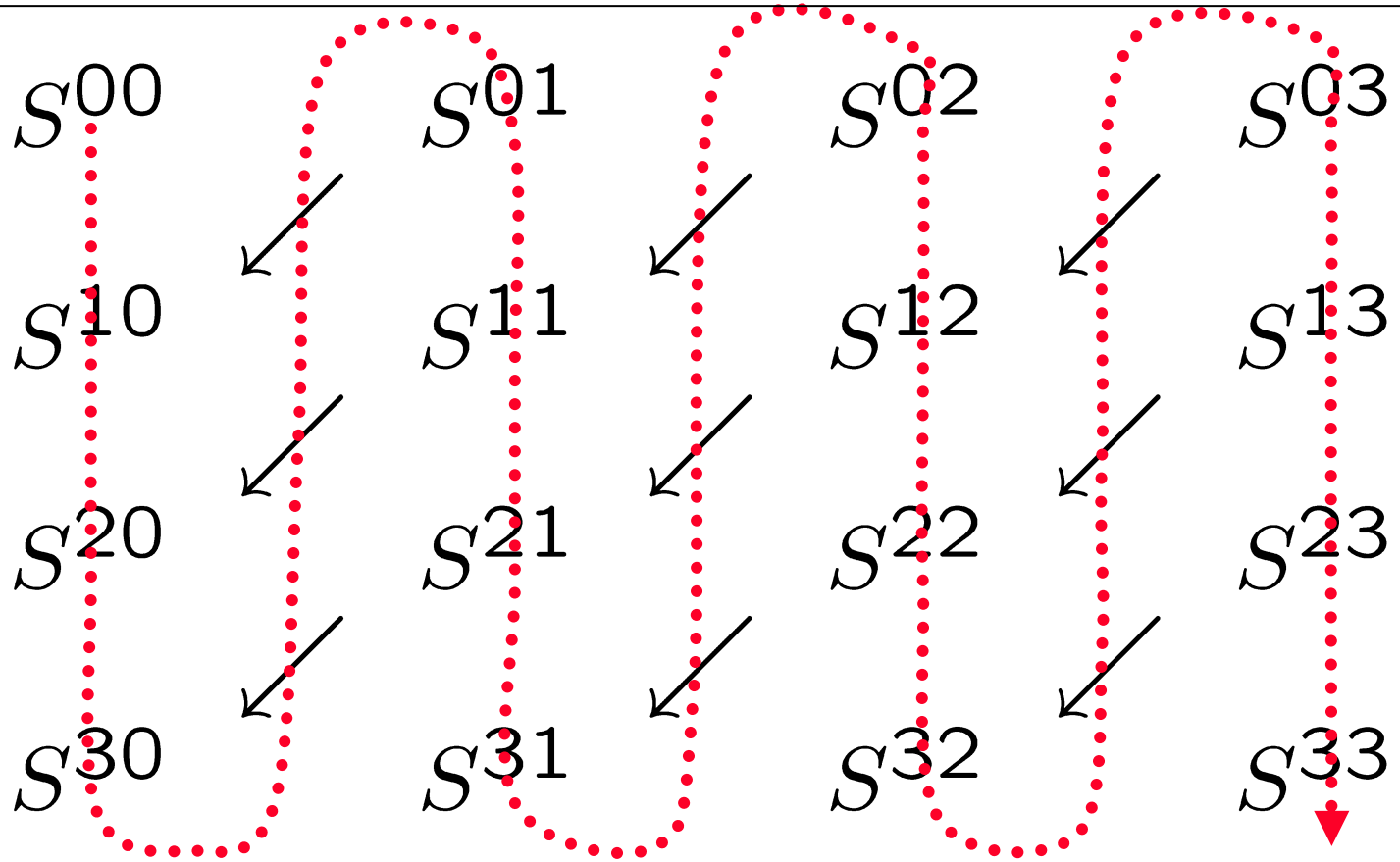


Interchange: counter-example

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
    A[I1,I2] := A[I1 - 1, I2 + 1] + B[I1, I2]
```

After interchange:

New traversal order crosses dependence arrows backwards



Interchange: condition

- **A loop is *interchangeable* if all dependence constraints (as shown by the arrows) are still satisfied by the top-to-bottom, left-to-right execution order**
- **How can you tell whether a loop can be interchanged?**

Look at its dependence direction vectors:

- **Is there a dependence direction vector with the form (<,>) ?**
  - **ie there is a dependence distance vector (k_1,k_2) with k_1>0 and k_2<0 ?**
  - **If so, interchange would be *invalid***
- **Because the arrows would be traversed backwards**
- **All other dependence directions are OK.**

Skewing

Consider this variation on the wavefront loop:

```
for k1 := 0 to 3 do
  for k2 := k1 to k1+3 do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

- The inner loop's control variable runs from k_1 to k_1+3.
- The iteration space of this loop has 4^2 iterations just like the original loop.
- If we draw the iteration space with each iteration S^{K_1,K_2} at coordinate position (K_1,K_2), it is skewed to form a lozenge shape:

$$
\begin{array}{cccccc}
S^{00} & S^{01} & S^{02} & S^{03} & & \\
 & S^{11} & S^{12} & S^{13} & S^{14} & \\
 & & S^{22} & S^{23} & S^{24} & S^{25} \\
 & & & S^{33} & S^{34} & S^{35} \quad S^{36}
\end{array}
$$

This loop performs the same computation as the original.

Skewing preserves semantics

```
for k1 := 0 to 3 do
  for k2 := k1 to k1+3 do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

S^{00} A_{00}	S^{01} A_{01}	S^{02} A_{02}	S^{03} A_{03}			
	S^{11} A_{10}	S^{12} A_{11}	S^{13} A_{12}	S^{14} A_{13}		
		S^{22} A_{20}	S^{23} A_{21}	S^{24} A_{22}	S^{25} A_{23}	
			S^{33} A_{30}	S^{34} A_{31}	S^{35} A_{32}	S^{36} A_{33}

- To see that this loop performs the same computation, lets work out its dependence structure.
- First label each iteration with the element of A to which it assigns
- The loop body is

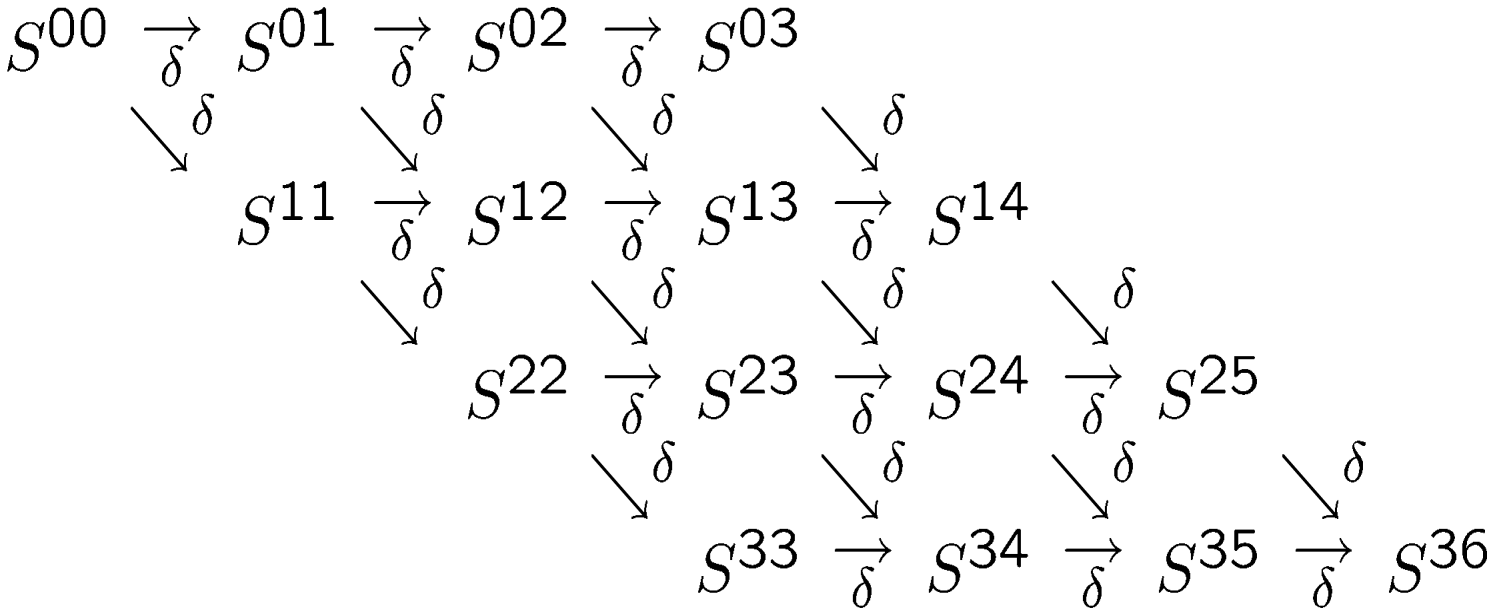
  $A[k_1,k_2-k_1] := A[k_1-1,k_2-k_1]+A[k_1,k_2-k_1-1]$

- E.g. iteration S_{23} does:

  $A[2,1] := A[1,1]+A[2,0]$

Skewing doesn't actually change the order in which the loop body is executed

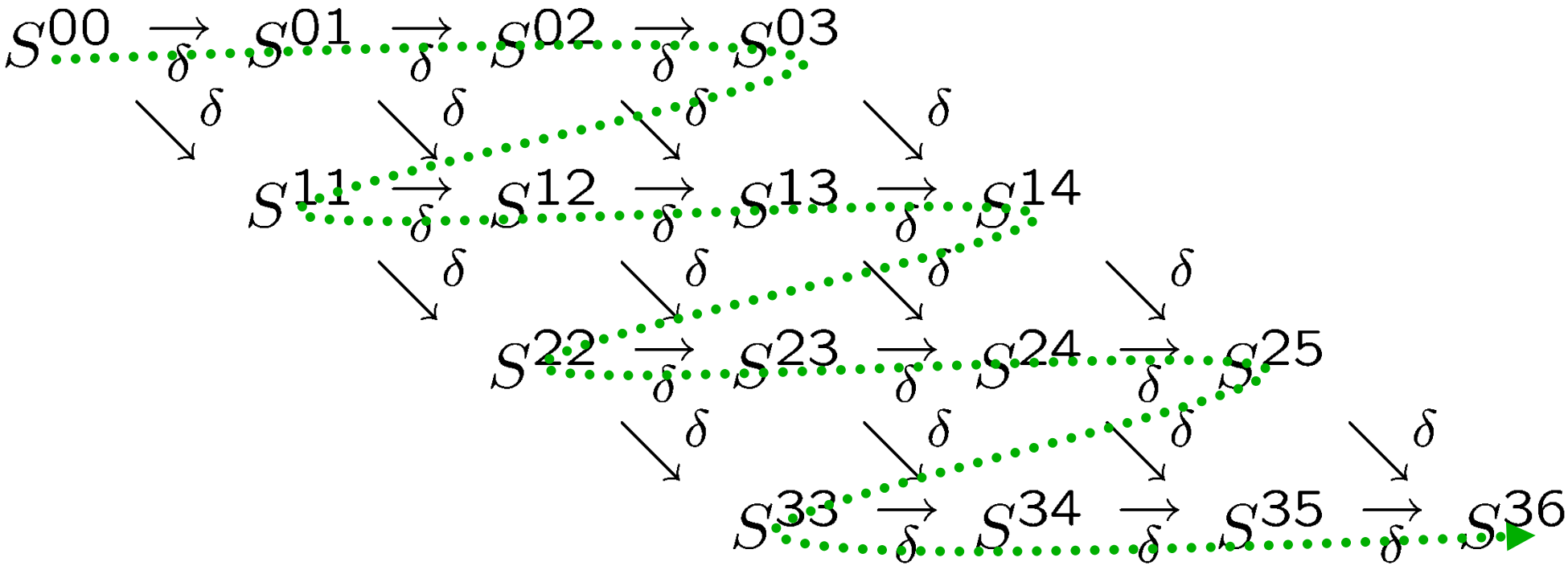
- Thus the dependence structure of the skewed loop is shown by marking the iteration space with all the dependences:



- Can this loop nest be vectorised?
- Can this loop nest be interchanged?

Skewing changes effect of interchange

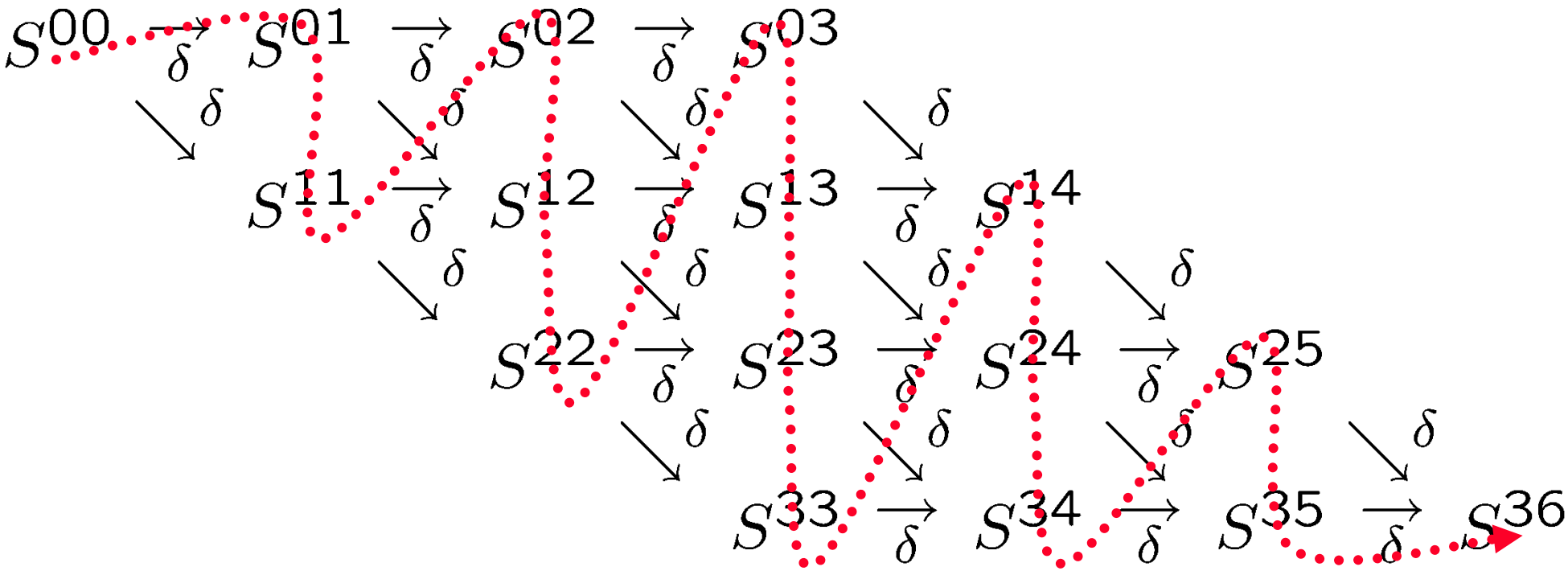
- Thus the dependence structure of the skewed loop is shown by marking the iteration space with all the dependences:



- Original execution order

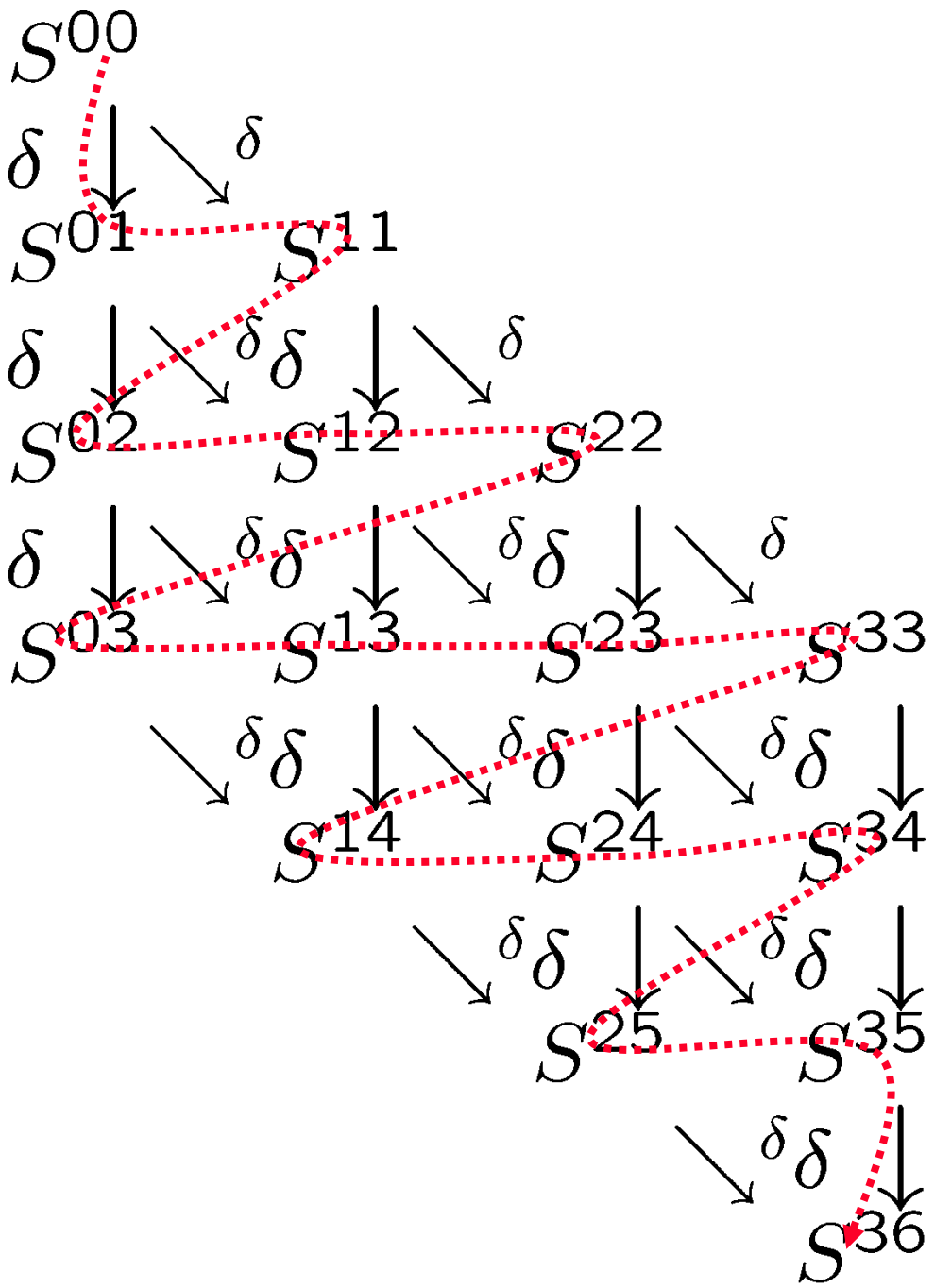
Interchange after skewing

- Thus the dependence structure of the skewed loop is shown by marking the iteration space with all the dependences:

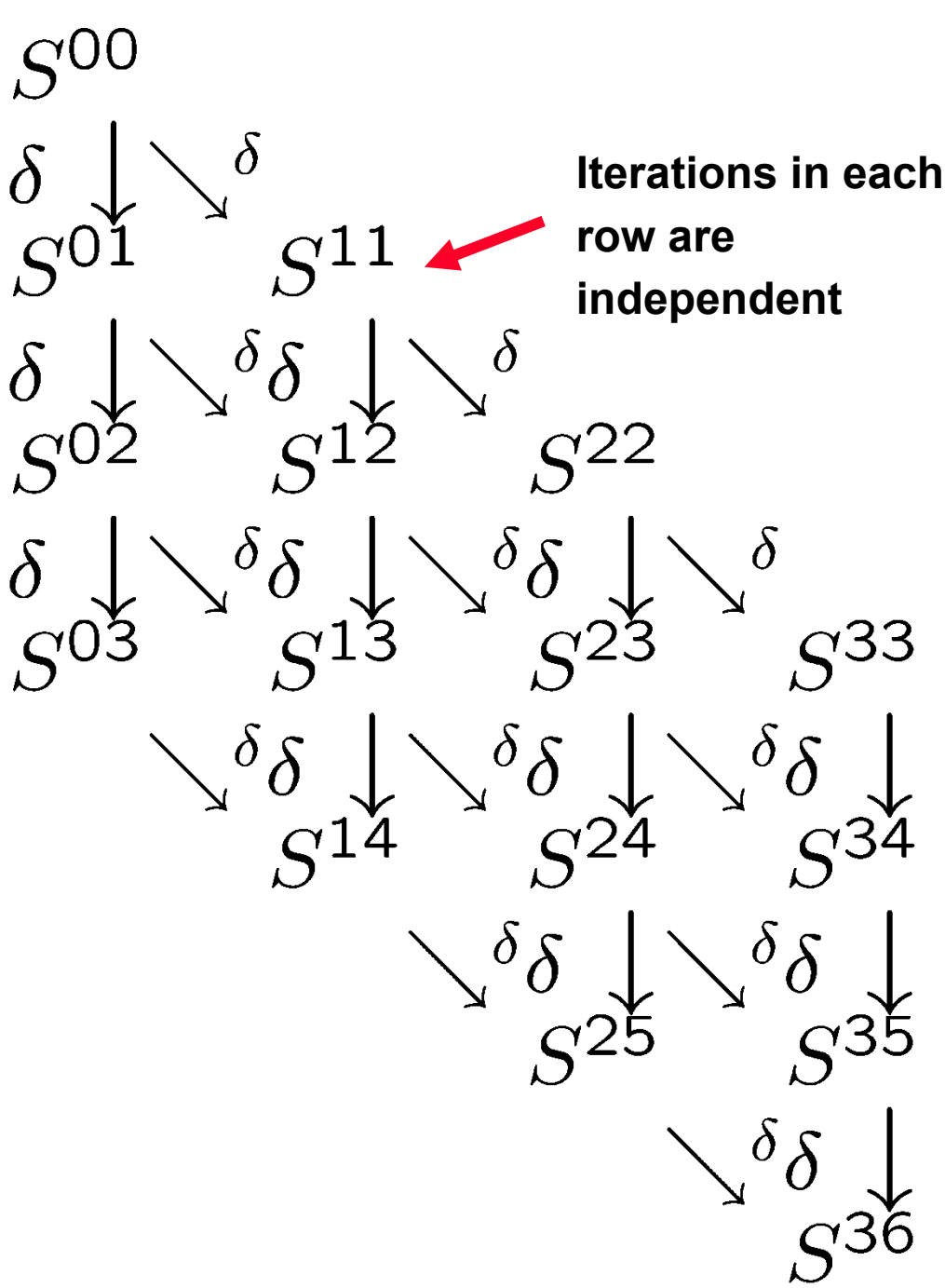


- Transposed execution order

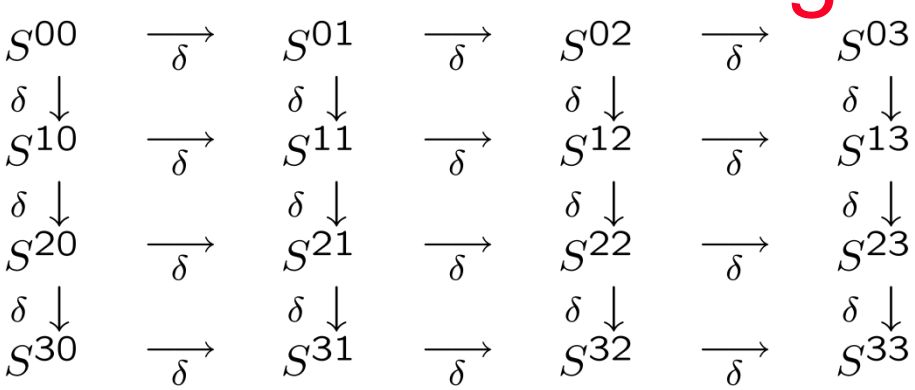
- You can think of loop interchange as changing the way the iteration space is traversed
- Alternatively, you can think of it as a change to the way the runtime code instances are mapped onto the iteration space
- Traversal is always lexicographic – ie left-to-right, top-down



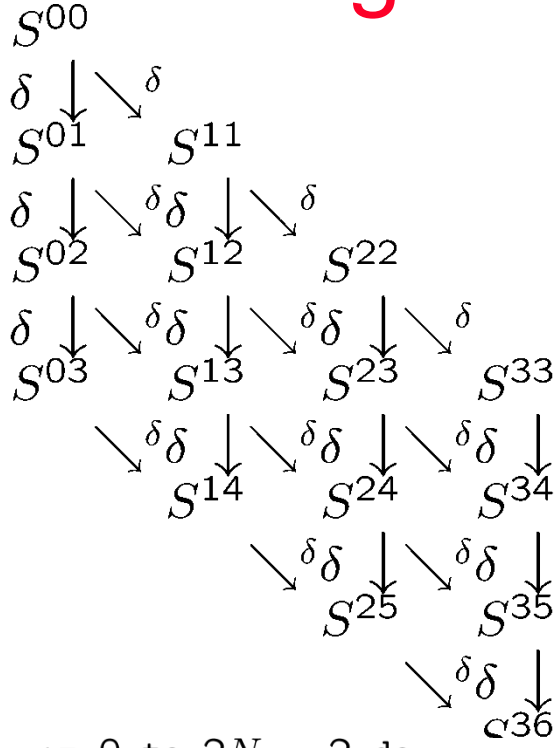
- The inner loop is now vectorisable, since it has no loop-carried dependence
- The skewed iteration space has N rows and 2N-1 columns, but still only N^2 actual statement instances.



Skewing and interchange: summary




```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S :   A[I1,I2] := A[I1 - 1,I2] + A[I1,I2 - 1]
```




```
for k2 := 0 to 2N2 - 2 do
  for k1 := max(0, K2 - N2 + 2) to min(K2, N1) do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

- **Original loop interchangeable but not vectorisable.**
- **We skewed inner loop by outer loop by factor 1.**
- **Still not vectorisable, but interchangeable.**
- **Interchanged, skewed loop *is* vectorisable.**
- **Bounds of new loop not simple!**

- **Is skewing ever invalid?**
- **Does skewing affect interchangeability?**
- **Does skewing affect dependence distances?**
- **Can you predict value of skewing?**

Summary: dependence

- Dependence equation for single loop:
  - **Suppose S_p refers to $A[\varphi_p(I)]$**
  - **Suppose S_q refers to $A[\varphi_q(I)]$**
  - **A dependence of some kind occurs between S_p and S_q if there exists a solution to the equation**

    $$\varphi_p(I^1) = \varphi_q(I^2)$$

  - **for integer values of I^1 and I^2 lying within the loop bounds**
- For multidimensional arrays, and nested for-loops, we generalise this to a system of simultaneous dependence equations for two iterations, (I_1^1, I_2^1) and (I_1^2, I_2^2)
- Iteration space graph**, lexicographic schedule of execution**
- **Arrows in graph show solutions to dependence equation**
- Dependence distance vectors **characterise families of congruent arrows**

Summary: transformations

- A loop can be executed in parallel **if it has no loop-carried dependence**
- A loop nest can be interchanged **if the transposed dependence distance vectors are lexicographically forward**
- Strip-mining **is always valid**
- Tiling **= strip-mining + interchange**

} Not explained yet

- **Skewing is always valid**
- Skewing can expose parallelism **by aligning parallel iterations with one of the loops**
- **Skewing can make interchange (and therefore tiling) valid**

Student question:

"why is antidependence a dependence?"

Dependence versus anti-dependence

- If the *uses* precede the *assignments*, we actually have an *anti*-dependence, ie

$$S_p \ \overline{\delta}_< \ S_q$$

if, for each solution pair $(\mathbf{I}^1, \mathbf{I}^2)$, $\mathbf{I}^1 > \mathbf{I}^2$

- In this case we do have a constraint on execution order
- Because we (may) have to read a value before it (may) be overwritten
- And this anti-dependence is loop-carried
- Anti-dependences prevent re-ordering, and multi-thread parallelism

"Loop-carried dependence"

- Consider this example:

```
for (int i=1; i<8; i++)
  c[i] = c[i-1] + b[i];
```

Q

Each iteration produces a value that is used in the next iteration

- When executed we get:

```
c[1] = c[0] + b[1];
c[2] = c[1] + b[2];
c[3] = c[2] + b[3];
c[4] = c[3] + b[4];
c[5] = c[4] + b[5];
c[6] = c[5] + b[6];
c[7] = c[6] + b[7];
c[8] = c[7] + b[8];
```

The dependence arrows go from one iteration to the next

The dependence is *carried* by the loop

Loop-carried true dependence:

```
for i
  A[i] = A[i-1] + B[i]
```

Loop-carried anti-dependence:

```
for i
  A[i] = A[i+1] + B[i]
```

“Loop-carried anti-dependence”

- Consider this example:

```
for (int i=0; i<7; i++)
  c[i] = c[i+1] + b[i];
```

- When executed we get:

```
c[0] = c[1] + b[1];
c[1] = c[2] + b[2];
c[2] = c[3] + b[3];
c[3] = c[4] + b[4];
c[4] = c[5] + b[5];
c[5] = c[6] + b[6];
c[6] = c[7] + b[7];
```

Each iteration uses a value which is overwritten in the next iteration

We need the use to happen before the overwrite

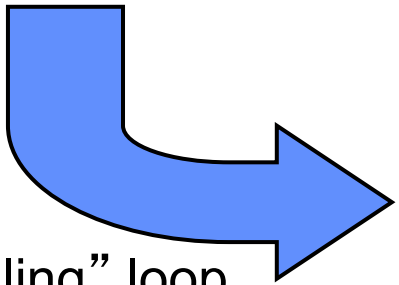
So we have a *precedence* requirement due to an anti-dependence

The anti-dependence arrows go from one iteration to the next

The anti-dependence is *carried* by the loop

Implementing shared-memory parallel loop

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



```
if (myThreadId() == 0)
  i = 0;
barrier();
// on each thread
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  A[local_i] = A[local_i] + B[local_i];
}
barrier();
```

Barrier(): block until all threads reach this point

- "self-scheduling" loop
- FetchAndAdd() is atomic operation to get next un-executed loop iteration:

```
Int FetchAndAdd(int *i) {
  lock(i);
  r = *i;
  *i = *i+1;
  unlock(i);
  return(r);
}
```

Optimisations:
- Work in chunks
- Avoid unnecessary barriers
- Exploit "cache affinity" from loop to loop

There are smarter ways to implement FetchAndAdd….

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

Thread #0

```
if (myThreadId() == 0)
  i = 0;
barrier();
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  A[local_i] = A[local_i] + B[local_i];
}
barrier();
```

Thread #1

```
if (myThreadId() == 0)
  i = 0;
barrier();
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  A[local_i] = A[local_i] + B[local_i];
}
barrier();
```

Thread #0 gets some sequence of iterations to do

Thread #1 gets some sequence of iterations to do

What could possibly go wrong?

```
for (i=0; i<N; i++) {
  A[i]  = A[i+1] + B[i];
}
```

Thread #0

```
if (myThreadId() == 0)
  i = 0;
barrier();
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  A[local_i] = A[local_i+1] + B[local_i];
}
barrier();
```

Thread #1

```
if (myThreadId() == 0)
  i = 0;
barrier();
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  A[local_i] = A[local_i+1] + B[local_i];
}
barrier();
```

Thread #0 gets some sequence of iterations to do

Thread #1 gets some sequence of iterations to do

```
for (i=0; i<N; i++) {
  A[i]  = A[i+1] + B[i];
}
```

What could possibly go wrong?

This example has a loop-carried anti-dependence.
We must read from A before overwriting A

Thread #0

```
if (myThreadId() == 0)
  i = 0;
barrier();
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  A[local_i] = A[local_i+1] + B[local_i];
}
barrier();
```

Thread #1

```
if (myThreadId() == 0)
  i = 0;
barrier();
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  A[local_i] = A[local_i+1] + B[local_i];
}
barrier();
```

Thread #0 gets some sequence of iterations to do, eg: **0, 2, 4, 6…**

Thread #1 gets some sequence of iterations to do, eg: **1, 3, 5, 7…**

Feeding curiosity: solving the dependence equation (not examinable)

```
from z3 import *
N=100
i1 = Int("i1")
i2 = Int("i2")
# consider a loop like this:
#  for i = 1 to N
#   a[phi1(i)] = a[phi2(i)] + b[i]
# So the dependence equation is
#   exists i1, i2: 1<i<n s.t. phi1(i1) == phi2(i2)
def DependenceTest(bounds, dependence_equation):
   s = Solver()
   s.add( bounds, dependence_equation )
   if s.check() == unsat:
      print ("No dependence is present")
   else:
      print("Dependence is found, for example when:")
      m = s.model()
      print ("i1 = %s (LHS)" % m[i1])
      print ("i2 = %s (RHS)" % m[i2])
```

Example 1:

```
print("for i = 1 to N")
print("  a[i] = a[i-1] + b[i]")
DependenceTest( And(i1>=1, i1<N, i2>=1, i2<N),
                i1 == i2-1 )
```

```
for i = 1 to N
  a[i] = a[i-1] + b[i]
Dependence is found, for example when:
i1 = 1 (LHS)
i2 = 2 (RHS)
```

Just add the constraints and call the solver

Feeding curiosity: solving the dependence equation

```
def DependenceTest(bounds, dependence_equation):
    s = Solver()
    s.add( bounds, dependence_equation )
    if s.check() == unsat:
        print ("No dependence is present")
    else:
        print("Dependence is found, for example when:")
        m = s.model()
        print ("i1 = %s (LHS)" % m[i1])
        print ("i2 = %s (RHS)" % m[i2])
        # Is there a loop-carried true dependence?
        s2 = Solver()
        s2.add( bounds, dependence_equation, i1<i2 )
        if s2.check() == unsat:
            print ("No loop-carried true dependence is present")
        else:
            print("Loop-carried true dependence found, for example when:")
            m = s2.model()
            print ("i1 = %s" % m[i1])
            print ("i2 = %s" % m[i2])
        # Is there a loop-carried anti-dependence?
        s3 = Solver()
        s3.add( bounds, dependence_equation, i1>i2 )
        if s3.check() == unsat:
            print ("No loop-carried anti-dependence is present")
        else:
            print("Loop-carried anti-dependence found, for example when:")
            m = s3.model()
            print ("i1 = %s" % m[i1])
            print ("i2 = %s" % m[i2])
```

Example 1:

```
print("for i = 1 to N")
print("  a[i] = a[i-1] + b[i]")
DependenceTest( And(i1>=1, i1<N, i2>=1, i2<N),
                i1 == i2-1 )
```

for i = 1 to N

a[i] = a[i-1] + b[i]

Dependence is found, for example when:

i1 = 1 (LHS)

i2 = 2 (RHS)

Loop-carried true dependence found, for example when:

i1 = 1

i2 = 2

No loop-carried anti-dependence is present

Extend to distinguish loop-carried true and anti-dependencies

Feeding curiosity: solving the dependence equation

```
def DependenceTest(bounds, dependence_equation):
    s = Solver()
    s.add( bounds, dependence_equation )
    if s.check() == unsat:
        print ("No dependence is present")
    else:
        print("Dependence is found, for example when:")
        m = s.model()
        print ("i1 = %s (LHS)" % m[i1])
        print ("i2 = %s (RHS)" % m[i2])
        # Is there a loop-carried true dependence?
        s2 = Solver()
        s2.add( bounds, dependence_equation, i1<i2 )
        if s2.check() == unsat:
            print ("No loop-carried true dependence is present")
        else:
            print("Loop-carried true dependence found, for example when:")
            m = s2.model()
            print ("i1 = %s" % m[i1])
            print ("i2 = %s" % m[i2])
        # Is there a loop-carried anti-dependence?
        s3 = Solver()
        s3.add( bounds, dependence_equation, i1>i2 )
        if s3.check() == unsat:
            print ("No loop-carried anti-dependence is present")
        else:
            print("Loop-carried anti-dependence found, for example when:")
            m = s3.model()
            print ("i1 = %s" % m[i1])
            print ("i2 = %s" % m[i2])
```

Example 2:

```
print("for i = 1 to N")
print("  a[i] = a[i] + b[i]")
DependenceTest( And(i1>=1, i1<N, i2>=1, i2<N),
                i1 == i2 )
```

```
for i = 1 to N
  a[i] = a[i] + b[i]
Dependence is found, for example when:
i1 = 1 (LHS)
i2 = 1 (RHS)
No loop-carried true dependence is present
No loop-carried anti-dependence is present
```

In this case the dependence is present but not loop-carried

Feeding curiosity: solving the dependence equation

```
def DependenceTest(bounds, dependence_equation):
    s = Solver()
    s.add( bounds, dependence_equation )
    if s.check() == unsat:
        print ("No dependence is present")
    else:
        print("Dependence is found, for example when:")
        m = s.model()
        print ("i1 = %s (LHS)" % m[i1])
        print ("i2 = %s (RHS)" % m[i2])
        # Is there a loop-carried true dependence?
        s2 = Solver()
        s2.add( bounds, dependence_equation, i1<i2 )
        if s2.check() == unsat:
            print ("No loop-carried true dependence is present")
        else:
            print("Loop-carried true dependence found, for example when:")
            m = s2.model()
            print ("i1 = %s" % m[i1])
            print ("i2 = %s" % m[i2])
        # Is there a loop-carried anti-dependence?
        s3 = Solver()
        s3.add( bounds, dependence_equation, i1>i2 )
        if s3.check() == unsat:
            print ("No loop-carried anti-dependence is present")
        else:
            print("Loop-carried anti-dependence found, for example when:")
            m = s3.model()
            print ("i1 = %s" % m[i1])
            print ("i2 = %s" % m[i2])
```

Example 3:

```
print("for i = 1 to N")
print("  a[2*i] = a[2*i-1] + b[i]")
DependenceTest( And(i1>=1, i1<N, i2>=1, i2<N),
                2*i1 == 2*i2-1 )
```

```
for i = 1 to N
  a[2*i] = a[2*i-1] + b[2*i]
No dependence is present
```

Feeding curiosity: solving the dependence equation

```
def DependenceTest(bounds, dependence_equation):
    s = Solver()
    s.add( bounds, dependence_equation )
    if s.check() == unsat:
        print ("No dependence is present")
    else:
        print("Dependence is found, for example when:")
        m = s.model()
        print ("i1 = %s (LHS)" % m[i1])
        print ("i2 = %s (RHS)" % m[i2])
        # Is there a loop-carried true dependence?
        s2 = Solver()
        s2.add( bounds, dependence_equation, i1<i2 )
        if s2.check() == unsat:
            print ("No loop-carried true dependence is present")
        else:
            print("Loop-carried true dependence found, for example when:")
            m = s2.model()
            print ("i1 = %s" % m[i1])
            print ("i2 = %s" % m[i2])
        # Is there a loop-carried anti-dependence?
        s3 = Solver()
        s3.add( bounds, dependence_equation, i1>i2 )
        if s3.check() == unsat:
            print ("No loop-carried anti-dependence is present")
        else:
            print("Loop-carried anti-dependence found, for example when:")
            m = s3.model()
            print ("i1 = %s" % m[i1])
            print ("i2 = %s" % m[i2])
```

Example 4:

```
print("for i = 1 to N")
print("  a[3*i] = a[5*i-10] + b[i]")
DependenceTest( And(i1>=1, i1<N, i2>=1, i2<N),
                3*i1 == 5*i2-20 )
```

for i = 1 to N

a[3*i] = a[5*1-20] + b[i]

Dependence is found, for example when:

i1 = 5 (LHS)

i2 = 7 (RHS)

Loop-carried true dependence found, for example when:

i1 = 5

i2 = 7

Loop-carried anti-dependence found, for example when:

i1 = 15

i2 = 13

In this case we have both true and anti-dependences: weird!

S1 : A[0] := 0

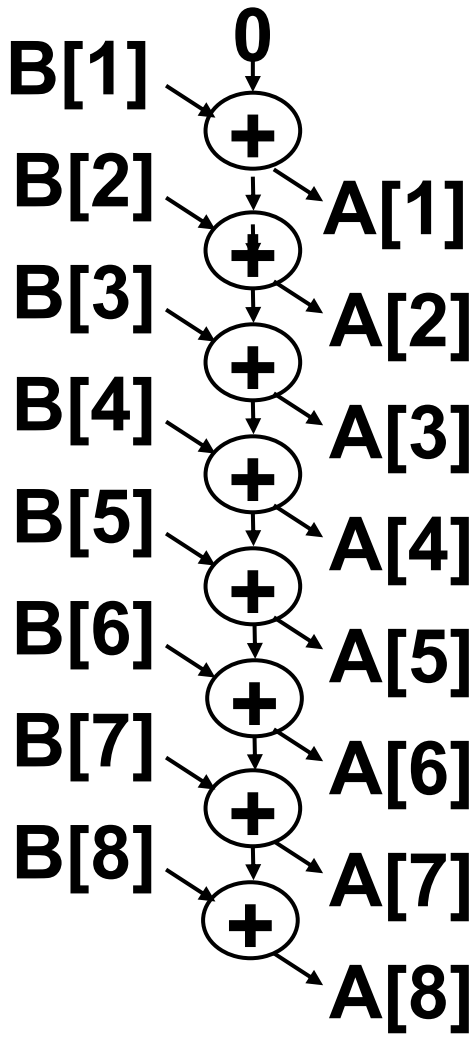
for i = 1 to 8

S2 : A[i] := A[i-1] + B[i]

Feeding curiosity (not examinable)

Loop-carried dependences can sometimes still be parallelised

- **Appears to be inherently sequential**

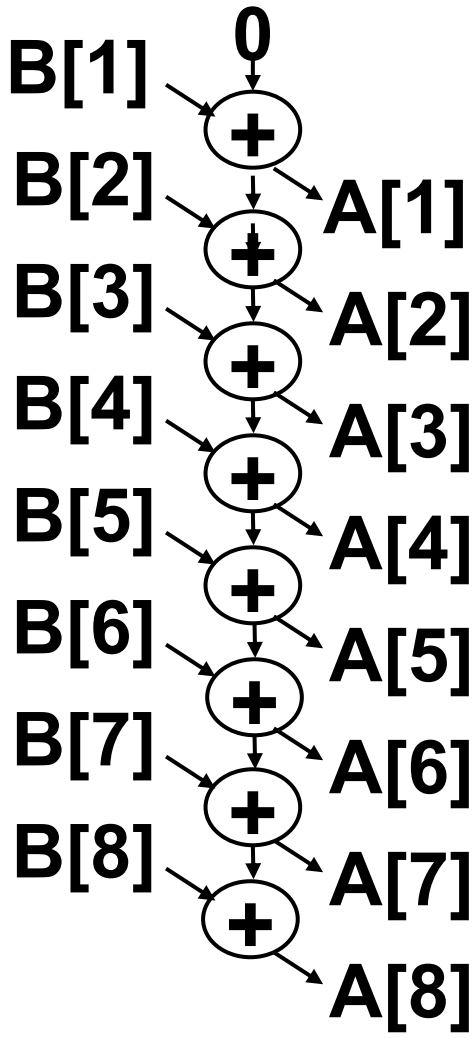


```
S1 :  A[0] := 0
      for i = 1 to 8
S2 :    A[i] := A[i-1] + B[i]
```

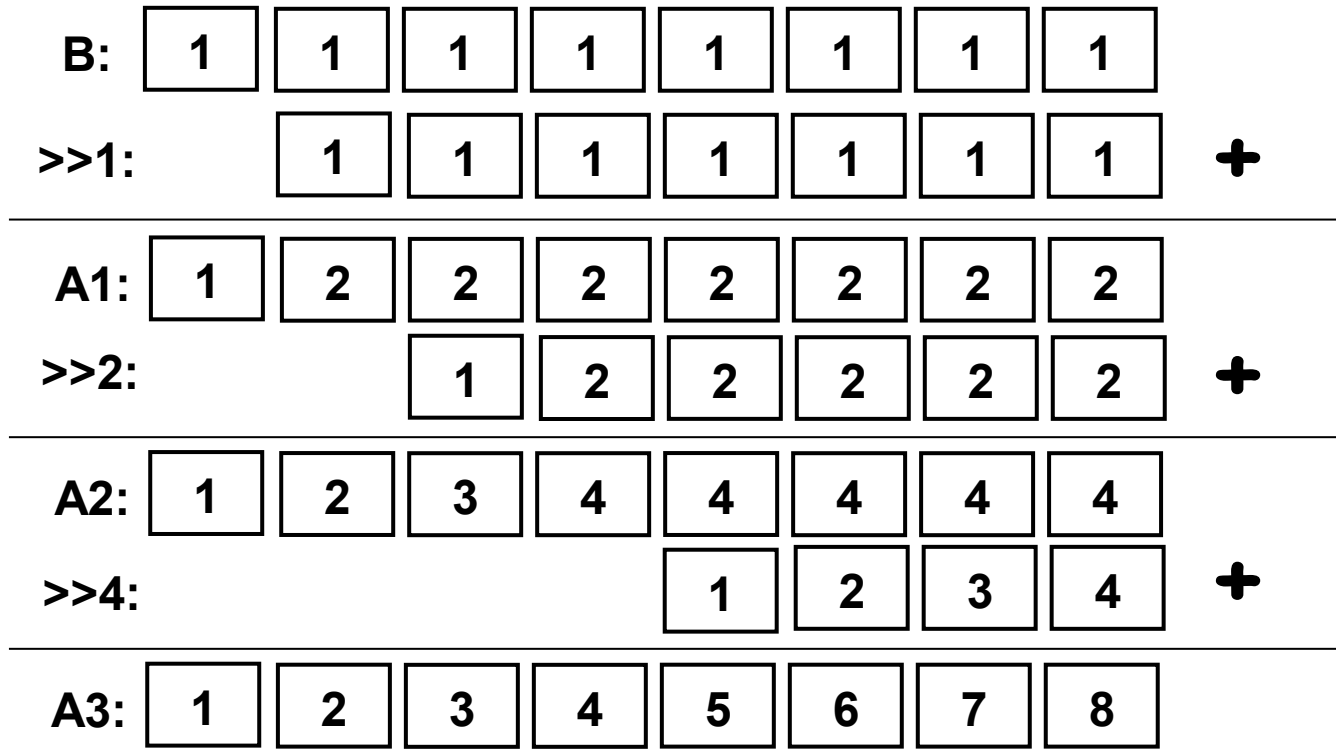
Feeding curiosity (not examinable)

Loop-carried dependences can sometimes still be parallelised

- **Appears to be inherently sequential**



- **But parallel is possible:**



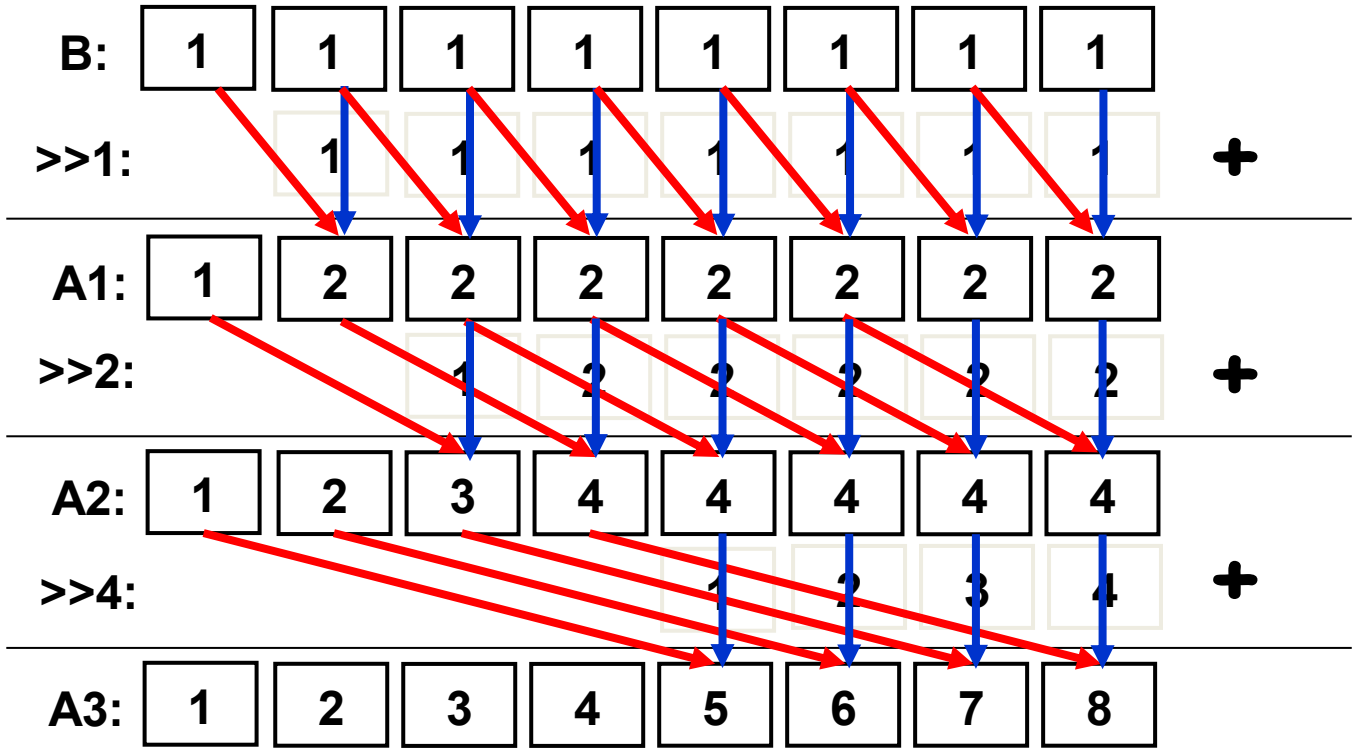
"Parallel scan" or "parallel prefix sum"

```
S1 :  A[0] := 0
      for i = 1 to 8
S2 :    A[i] := A[i-1] + B[i]
```

Feeding curiosity (not examinable)

Loop-carried dependences can sometimes still be parallelised

- **Appears to be inherently sequential**
- **But parallel implementation is possible**



- **Each step is a vector-parallel operation**
- **Of decreasing size**
- **We have log(N) steps**

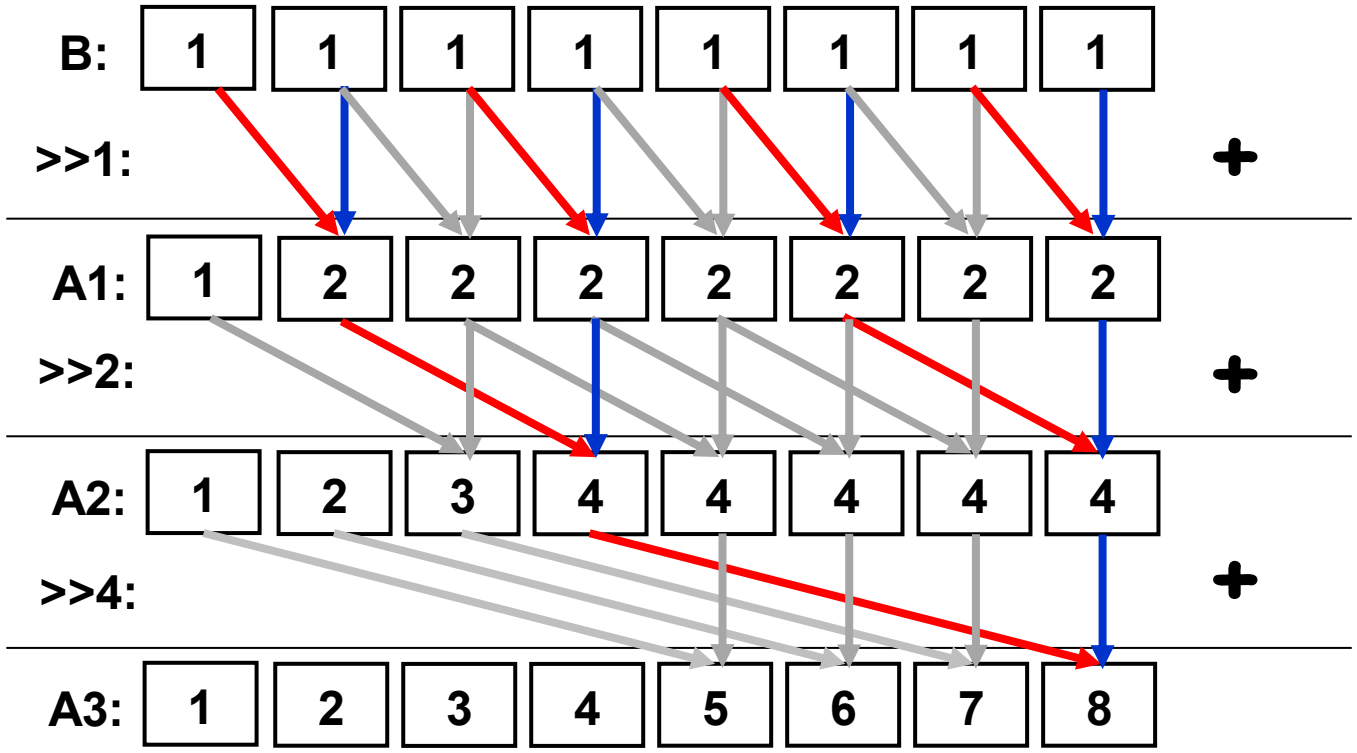
"Parallel scan" or "parallel prefix sum"

```
S1 :  A[0] := 0
      for i = 1 to 8
S2 :    A[i] := A[i-1] + B[i]
```

Feeding curiosity (not examinable)

Loop-carried dependences can sometimes still be parallelised

- **Appears to be inherently sequential**
- **But parallel implementation is possible**



We can see that the last element is computed with a reduction tree

S1 : A[0] := 0

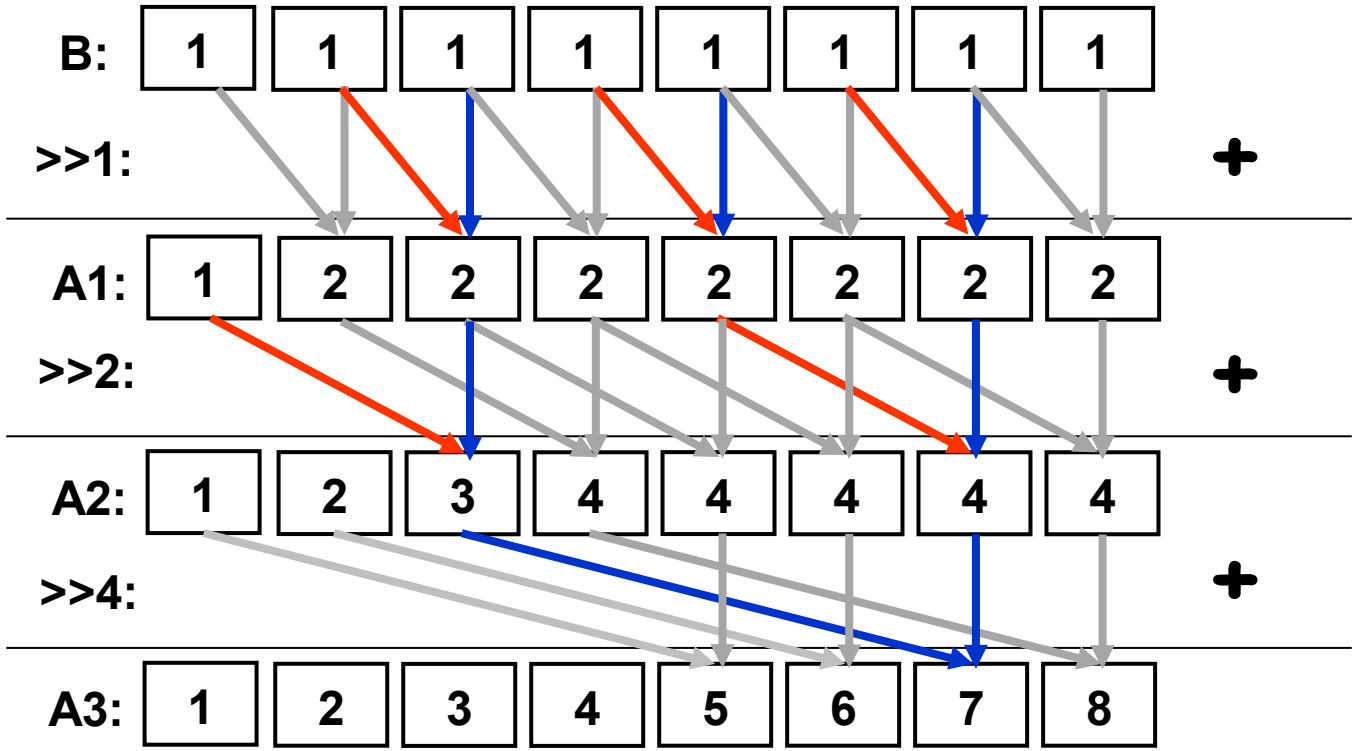
for i = 1 to 8

S2 : A[i] := A[i-1] + B[i]

Feeding curiosity (not examinable)

Loop-carried dependences can sometimes still be parallelised

- **Appears to be inherently sequential**
- **But parallel implementation is possible**

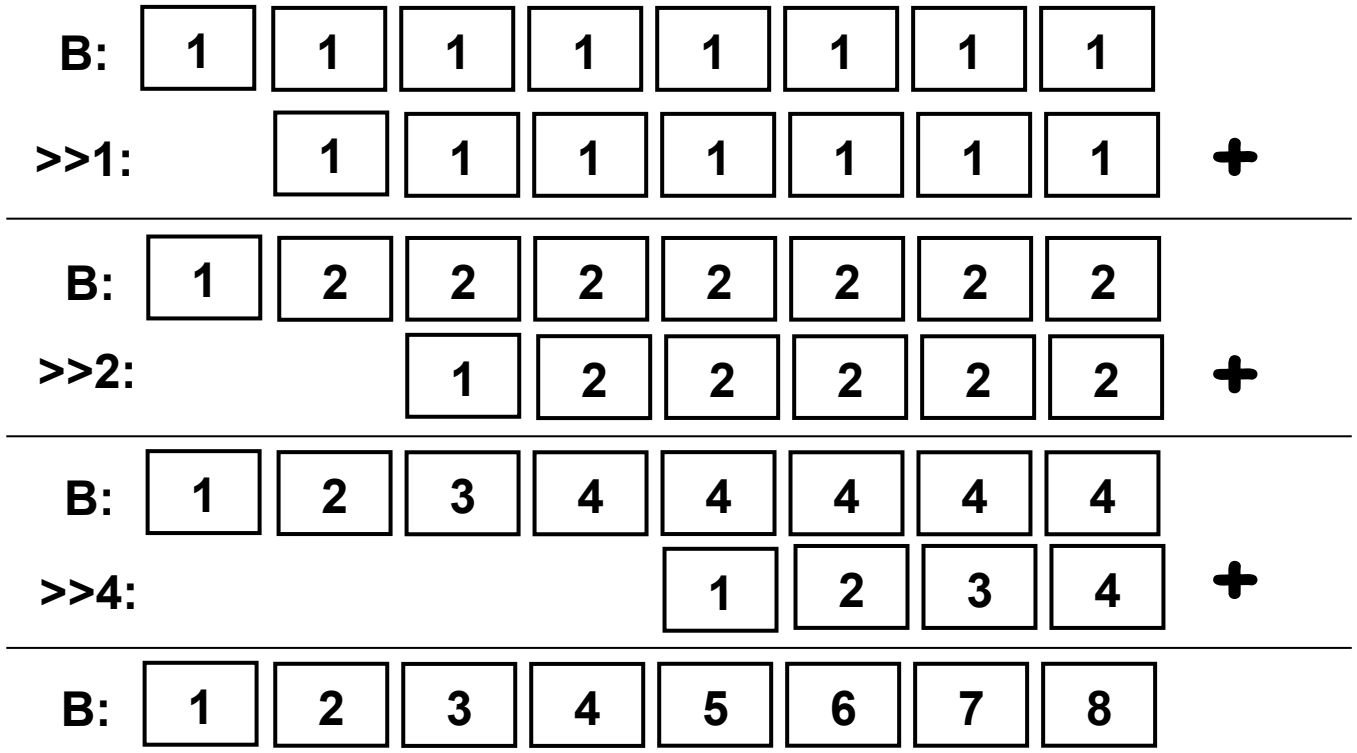


All the elements are computed by reduction trees of depth log(N) – for example element 7

Feeding curiosity

```
S1 :  A[0] := 0
      for i = 1 to 8
S2 :    A[i] := A[i-1] + B[i]
```

- Appears to be inherently sequential
- But parallel implementation is possible



"Parallel scan" or "parallel prefix sum"

- This is the "naïve" parallel scan
- It does more work than the sequential scan – but it does use parallelism
- There are "work-efficient" parallel scans
- Eg see Mark Harris, GPU Gems Ch39

https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

Compilers - That wraps it up!

We have seen….

- How to build a simple non-optimising compiler for a simple imperative language
- With functions
- With local variables, static variables, heap data, inheritance
- We have seen how an optimising compiler might work –
  - intermediate representations, lowering,
  - dataflow analysis, register allocation, code motion optimisations,
  - instruction selection
  - SSA
- Dependence analysis and parallelisation
  - Loop-carried dependence
  - Dependence distance
  - Vectorisation and parallelisation

- You have been introduced to a world of topics fundamental to how your code actually gets executed, and what can be done to make it efficient

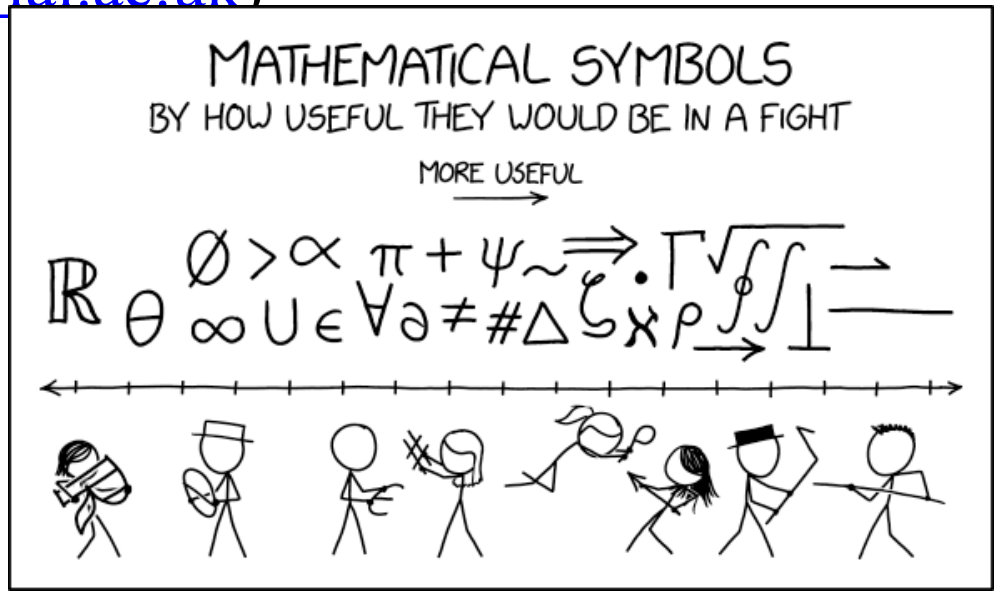
Compilers - Chapter 8:

Loop scheduling optimisations

Part 4: Representing loop transformations as matrix multiplications

- Lecturer:
  - Paul Kelly (p.kelly@imperial.ac.uk)

This section is not examinable



https://xkcd.com/2343/

Matrix representation of loop transformations

- **To skew the inner loop by the outer loop by factor 1 we adjust the loop bounds, and replace I_1 by K_1, and I_2 by K_2-K_1. That is,**

  $$(K_1,K_2) = (I_1,I_2) \cdot U$$

- **where U is a 2 x 2 matrix**

$$\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

- **That is,**

  $$(K_1,K_2) = (I_1,I_2) \cdot U = (I_1,I_2+I_1)$$

The matrix representation is not examinable

Matrix representation of loop transformations

- To skew the inner loop by the outer loop by factor 1 we adjust the loop bounds, and replace I_1 by K_1, and I_2 by K_2-K_1. That is,

  $(K_1,K_2) = (I_1,I_2) \cdot U$

- where U is a 2 x 2 matrix

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{U^{-1}} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

- That is,

  $(K_1,K_2) = (I_1,I_2) \cdot U = (I_1,I_2+I_1)$

- The inverse gets us back again:

  $(I_1,I_2) = (K_1,K_2) \cdot U^{-1} = (K_1,K_2-K_1)$

The matrix representation is not examinable

- Matrix U maps each statement instance $S^{I_1 I_2}$ to its position in the new iteration space, $S^{K_1 K_2}$:

- Original iteration space:

I_1 \ I_2 :	0	1	2	3
0	S^{00}	S^{01}	S^{02}	S^{03}
1	S^{10}	S^{11}	S^{12}	S^{13}
2	S^{20}	S^{21}	S^{22}	S^{23}
3	S^{30}	S^{31}	S^{32}	S^{33}

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S :   A[I1,I2] := A[I1 − 1,I2] + A[I1,I2 − 1]

for k1 := 0 to 3 do
  for k2 := k1 to k1+3 do
S : A[k1,k2−k1] := A[k1−1,k2−k1]+A[k1,k2−k1−1]
```

The subscripts are mapped back using U^{-1}

-

K_1 \ K_2 :	0	1	2	3	4	5	6
0	S^{00}	S^{01}	S^{02}	S^{03}			
1		S^{11}	S^{12}	S^{13}	S^{14}		
2			S^{22}	S^{23}	S^{24}	S^{25}	
3				S^{33}	S^{34}	S^{35}	S^{36}

The dependences are subject to the same transformation.

$(K_1,K_2) = (I_1,I_2) \cdot U = (I_1,I_2+I_1)$

$(I_1,I_2) = (K_1,K_2) \cdot U^{-1} = (K_1,K_2-K_1)$

The matrix representation is not examinable

Using matrices to reason about dependence

Recall that:

- There is a dependence between two iterations (I_1^1, I_2^1) and (I_1^2, I_2^2) if there is a memory location which is assigned to in iteration (I_1^1, I_2^1), and read in iteration (I_1^2, I_2^2).

  ((unless there is an intervening assignment))

- If (I_1^1, I_2^1) precedes (I_1^2, I_2^2) it is a *data*-dependence.
- If (I_1^2, I_2^2) precedes (I_1^1, I_2^1) it is a *anti*-dependence.
- If the location is assigned to in both iterations, it is an *output*-dependence.

- The dependence distance vector (D_1, D_2) is $(I_1^1 - I_1^2, I_2^1 - I_2^2)$.

The matrix representation is not examinable

Transforming dependence vectors

- **If there is a dependence between two iterations (I_1^1, I_2^1) and (I_1^2, I_2^2)**
- **Then iterations (I_1^1, I_2^1) . U and (I_1^2, I_2^2) .U will also read and write the same location**

- **The transformation U is *valid* iff**

  (I_1^1, I_2^1) . U precedes (I_1^2, I_2^2) . U

  whenever there is a dependence between

  (I_1^1, I_2^1) and (I_1^2, I_2^2).

- **In the transformed loop the dependence distance vector is also transformed, to**

  (D_1, D_2) . U

- **U is a valid transformation if all the program's dependence distance vectors are still "forward" when transformed by U**

The matrix representation is not examinable

Transforming dependence vectors

- What do we mean by "precedes"?

Definition: Lexicographic ordering:

(I^1,J^1) precedes (I^2,J^2)

if $I^1 < I^2$, or $I^1 = I^2$ and $J^1 < J^2$

- "Lexicographic" is dictionary order – both "baz" and "can" precede "cat"
- So (1,2) precedes (1,3)
- But (0,3) precedes (1,4)
- A dependence distance vector (D_1,D_2) is lexicographically "forward" if it precedes (0,0)

The matrix representation is not examinable

Example: loop given earlier

Before transformation we had two dependences:

1. Distance: (1,0), direction: (<,.)
2. Distance: (0,1), direction: (.,<)

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S :   A[I1,I2] := A[I1 − 1,I2] + A[I1,I2 − 1]
```

```
for k1 := 0 to 3 do
  for k2 := k1 to k1+3 do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

- After transformation by matrix

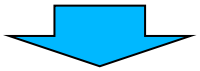
$$\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

- (i.e. skewing of inner loop by outer) we get:

1. Distance: (1,0).U = (1,1), direction: (<,<)
2. Distance: (0,1).U = (0,1), direction: (.,<)

I_1 \ I_2 :	0	1	2	3
0	S^{00}	S^{01}	S^{02}	S^{03}
1	S^{10}	S^{11}	S^{12}	S^{13}
2	S^{20}	S^{21}	S^{22}	S^{23}
3	S^{30}	S^{31}	S^{32}	S^{33}

→ (0,1)
↓ (1,0)



K_1 \ K_2 :	0	1	2	3	4	5	6
0	S^{00}	S^{01}	S^{02}	S^{03}			
1		S^{11}	S^{12}	S^{13}	S^{14}		
2			S^{22}	S^{23}	S^{24}	S^{25}	
3				S^{33}	S^{34}	S^{35}	S^{36}

→ (0,1)
↘ (1,1)

The matrix representation is not examinable

- We can also represent loop interchange by a matrix transformation.
- After transforming the skewed loop by matrix

$$\mathbf{V} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- (i.e. loop interchange) we get:

1. Distance: (1,0).U.V = (1,1).V = (1,1), direction: (<,<)
2. Distance: (0,1).U.V = (0,1).V = (1,0), direction: (<,.)

- The transformed iteration space is the transpose of the skewed iteration space:

$$\begin{array}{cccc} S^{00} & & & \\ S^{10} & S^{11} & & \\ S^{20} & S^{21} & S^{22} & \\ S^{30} & S^{31} & S^{32} & S^{33} \\ & S^{41} & S^{42} & S^{43} \\ & & S^{52} & S^{53} \\ & & & S^{63} \end{array}$$

The matrix representation is not examinable

Summary

- (I_1,I_2) . U maps each statement instance (I_1,I_2) to its new position (K_1,K_2) in the transformed loop's execution sequence
- (D_1,D_2) . U gives new dependence distance vector, giving test for validity

- Captures skewing, interchange and reversal
- Compose transformations by matrix multiplication

  $U_1 . U_2$

- Resulting loop's bounds may be a little tricky
  - Efficient algorithms exist [Banerjee90] to maximise parallelism by skewing and loop interchanging
  - Efficient algorithms exist to optimise cache performance by finding the combination of blocking, block size, interchange and skewing which leads to the best reuse [Wolf91]

The matrix representation is not examinable

Restructuring compilers - conclusions:

- Restructuring compilers can find parallelism
- And enhance locality
- **For a very restricted class of programs**
  - For-loops over arrays with array subscripts that are simple ("affine") expressions involving loop control variables

- But for this restricted class there is a rather elegant theory (the "polyhedral" or "polytope" model, http://en.wikipedia.org/wiki/Polytope_model)
- Extending beyond this is a big research problem

- Current compilers (GCC, Clang/LLVM, Intel, Microsoft etc) can do some of this, in theory – but are often defeated by program complexity

References

- Textbooks covering restructuring compilers
  - Michael Wolfe. High Performance Compilers for Parallel Computing. Addison Wesley, 1996.
  - Steven Muchnick, Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.
  - Ken Kennedy and Randy Allen, Optimizing Compilers for Modern Architectures. Morgan Kaufmann, 2001.
- Research papers:
  - D. F. Bacon and S. L. Graham and O. J. Sharp, "Compiler Transformations for High-Performance Computing". ACM Computing Surveys V26 N4 Dec 1994 http://doi.acm.org/10.1145/197405.197406
  - U. Banerjee. Unimodular transformations of double loops. In Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, CA. Pitman/MIT Press, 1990.
  - M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, volume 26, pages 30-44, Toronto, Ontario, Canada, June 1991.
- Polyhedral:

The Polyhedral Model (aka Polytope method) takes the ideas in this lecture much further, notably

- Automatic code generation using the matrix model shown here
- Checking validity of such transformations
- Calculating parallelism and locality metrics for alternative versions
  - https://en.wikipedia.org/wiki/Polytope_model
  - https://normrubin.github.io/lectures/poly_final.html
  - https://polyhedral.info/
  - https://www.impact-workshop.org/

Matrix transpose

Feeding curiosity

Try this link to the the Compiler Explorer:

https://godbolt.org/#g:!((g:!((g:!((h:codeEditor,i:(filename:'1',fontScale:14,fontUsePx:'0',j:1,lang:c%2B%2B,selection:(endColumn:2,endLineNumber:20,positionColumn:2,positionLineNumber:20,selectionStartColumn:2,selectionStartLineNumber:20,startColumn:2,startLineNumber:20),source:'%23define+SIZE+10240%0A//%23define+SIZE+20480%0A%23define+TOTALBYTES+SIZE*SIZE*4%0A%0Aint+A%5BSIZE%5D%5BSIZE%5D%3B%0Aint+B%5BSIZE%5D%5BSIZE%5D%3B%0A%0A%23define+IB+32%0A%23define+JB+32%0A%0Avoid+P(int+N,+int+M)%0A%7B%0A++int+i,+j%3B%0A%0A++for+(i%3D0%3B+i%3CN%3B+i%2B%2B)+%7B%0A++++for+(j%3D0%3B+j%3CN%3B+j%2B%2B)+%7B%0A++++++B%5Bi%5D%5Bj%5D+%3D+A%5Bj%5D%5Bi%5D%3B%0A++++%7D%0A++%7D%0A%7D'),l:'5',n:'0',o:'C%2B%2B+source+%231',t:'0')),k:50,l:'4',n:'0',o:'',s:0,t:'0'),(g:!((h:compiler,i:(compiler:clang_trunk,filters:(b:'0',binary:'1',binaryObject:'1',commentOnly:'0',debugCalls:'1',demangle:'0',directives:'0',execute:'1',intel:'0',libraryCode:'0',trim:'1',verboseDemangling:'0'),flagsViewOpen:'1',fontScale:14,fontUsePx:'0',j:1,lang:c%2B%2B,libs:!(),options:'-Ofast+-march%3Dznver4',overrides:!(),selection:(endColumn:1,endLineNumber:1,positionColumn:1,positionLineNumber:1,selectionStartColumn:1,selectionStartLineNumber:1,startColumn:1,startLineNumber:1),source:1),l:'5',n:'0',o:'+x86-64+clang+(trunk)+(Editor+%231)',t:'0')),k:50,l:'4',n:'0',o:'',s:0,t:'0')),l:'2',n:'0',o:'',t:'0')),version:4

Collision detect

Try this link to the the Compiler Explorer:

https://godbolt.org/#g:!((g:!((g:!((h:codeEditor,i:(filename:'1',fontScale:14,fontUsePx:'0',j:1,lang:c%2B%2B,selection:(endColumn:13,endLineNumber:18,positionColumn:13,positionLineNumber:18,selectionStartColumn:13,selectionStartLineNumber:18,startColumn:13,startLineNumber:18),source:'%23define+SIZE+10240%0A//%23define+SIZE+20480%0A%23define+TOTALBYTES+SIZE*SIZE*4%0A%0Aint+A%5BSIZE%5D%3B%0Aint+B%5BSIZE%5D%3B%0Aint+C%5BSIZE%5D%3B%0Aint+D%5BSIZE%5D%3B%0A%0A%23define+IB+32%0A%23define+JB+32%0A%0Avoid+P(int+N,+int+M)%0A%7B%0A++int+i,+j%3B%0A%0A++for+(i%3D0%3B+i%3CN%3B+i%2B%2B)+%7B%0A++++++C%5BB%5Bi%5D%5D+%2B%3D+A%5BB%5Bi%5D%5D+%2B+D%5Bi%5D%3B%0A++%7D%0A%7D'),l:'5',n:'0',o:'C%2B%2B+source+%231',t:'0')),k:33.333333333333336,l:'4',n:'0',o:'',s:0,t:'0'),(g:!((h:compiler,i:(compiler:icxlatest,filters:(b:'0',binary:'1',binaryObject:'1',commentOnly:'0',debugCalls:'1',demangle:'0',directives:'0',execute:'1',intel:'0',libraryCode:'0',trim:'1',verboseDemangling:'0'),flagsViewOpen:'1',fontScale:14,fontUsePx:'0',j:1,lang:c%2B%2B,libs:!(),options:'-Ofast+-march%3Dznver4+',overrides:!(),selection:(endColumn:1,endLineNumber:1,positionColumn:1,positionLineNumber:1,selectionStartColumn:1,selectionStartLineNumber:1,startColumn:1,startLineNumber:1),source:1),l:'5',n:'0',o:'+x86-64+icx+2025.0.0+(Editor+%231)',t:'0')),k:33.333333333333336,l:'4',n:'0',o:'',s:0,t:'0'),(g:!((h:output,i:(compilerName:'x86-64+clang+(trunk)',editorid:1,fontScale:14,fontUsePx:'0',j:1,wrap:'1'),l:'5',n:'0',o:'Output+of+x86-64+icx+2025.0.0+(Compiler+%231)',t:'0')),k:33.33333333333333,l:'4',n:'0',o:'',s:0,t:'0')),l:'2',n:'0',o:'',t:'0')),version:4

Feeding curiosity

Ask me about….

- **Loop interchange for locality**
  - For i, j, k matrix multiply vs
  - For i, k, j matrix multiply
  - See **/homes/phjk/ToyPrograms/ACA24-25/MM** – compare speed of versions MM1.c, MM2.c
- **Tiling for locality**
  - For the transpose example shown in the last chapter
  - For matrix multiply (see version **MM3.c**)
- **Stencils and convolutions**
  - skewed, split, diamond
- **Graphs and unstructured meshes**