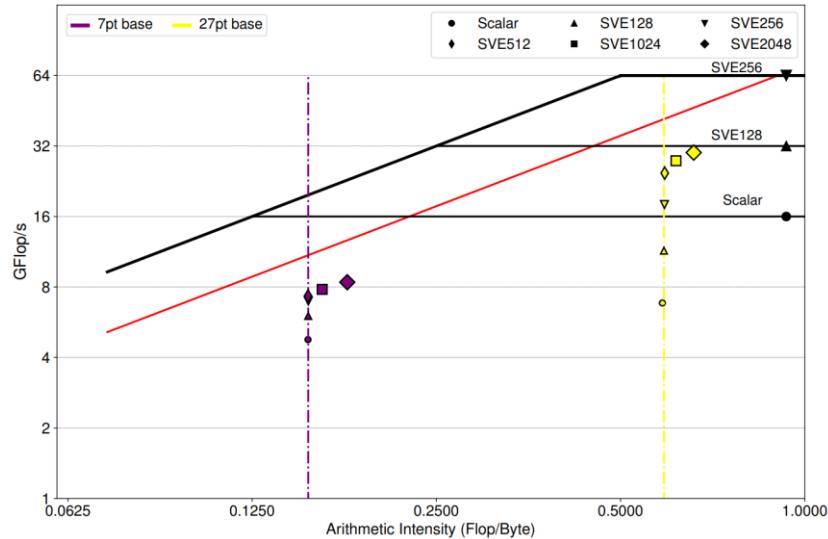


Advanced Computer Architecture

Chapter 8:

Vectors, vector instructions, vectorization and SIMD



November 2025
Paul H J Kelly

This section has contributions from Fabio Lupo (PhD & postdoc at Imperial, now CTO of DevitoCodes) and Luigi Nardi (ex Imperial and Stanford postdoc, now an academic at Lund University and founder at <https://www.dbtune.com/>).

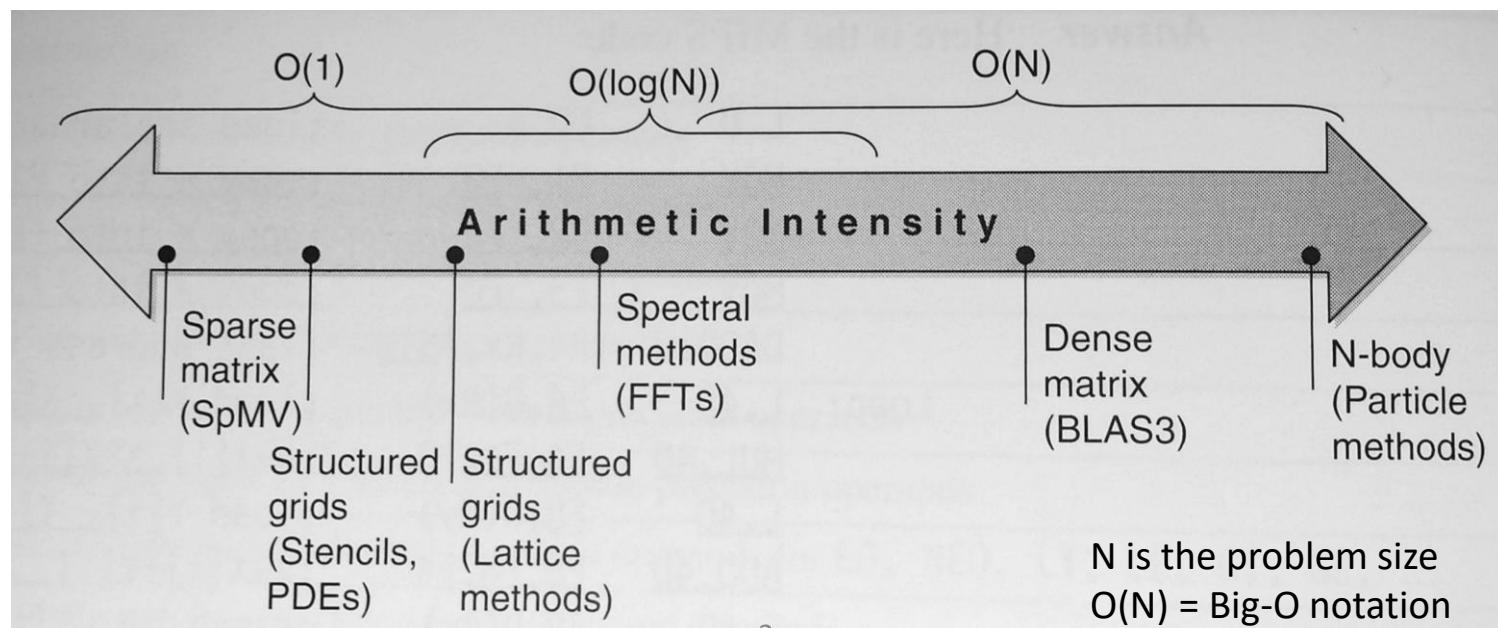
- ▶ Reducing Turing Tax
- ▶ Increasing instruction-level parallelism
- ▶ Roofline model: when does it matter?
- ▶ Vector instruction sets
 - ▶ Automatic vectorization (and what stops it from working)
 - ▶ How to make vectorization happen
- ▶ Lane-wise predication
- ▶ How are vector instructions actually executed?
- ▶ And then, in the next chapter: GPUs, and Single-Instruction Multiple Threads (SIMT)

Arithmetic Intensity

Processor	Type	Peak GFLOP/s	Peak GB/s	Ops/Byte	Ops/Word
Intel	E5-2690 v3* SP	CPU	416	68	~6
	E5-2690 v3 DP	CPU	208	68	~3
NVIDIA	K40** SP	GPU	4,290	288	~15
	K40 DP	GPU	1,430	288	~5

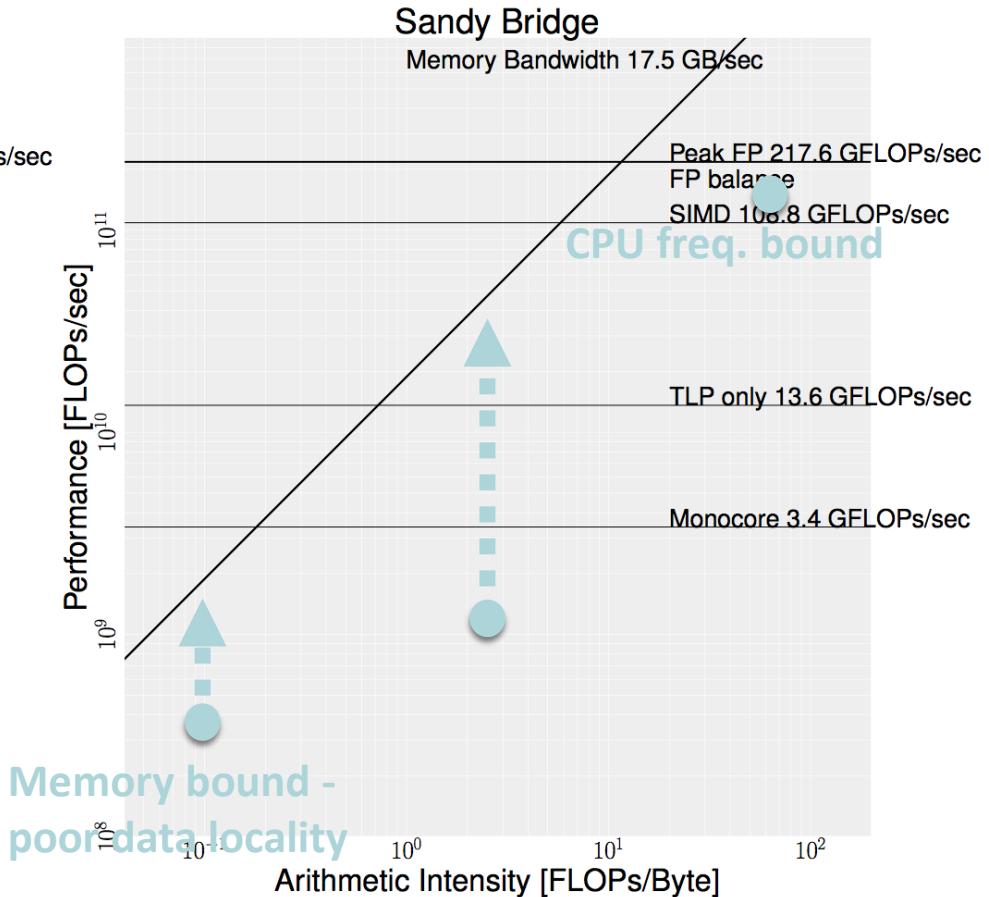
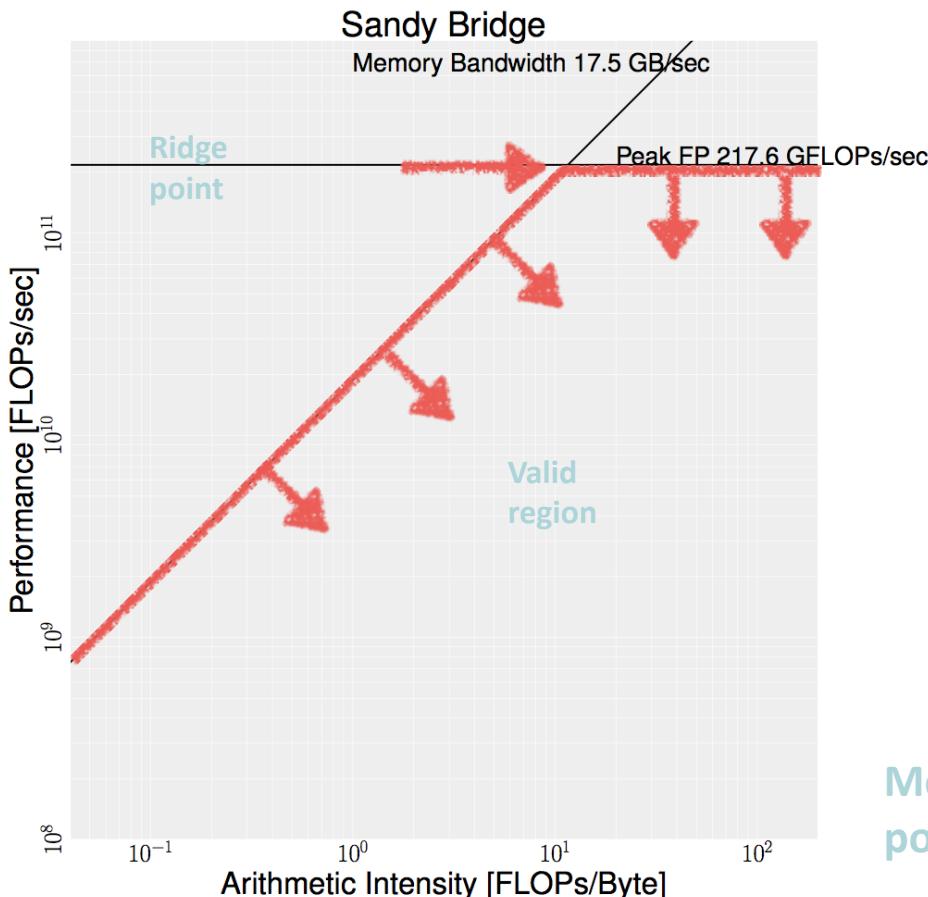
If the hardware has high Ops/Word, some code is likely to be bound by operand delivery
 (SP: single-precision, 4B/word; DP: double-precision, 8B/word)

Arithmetic intensity: Ops/Byte of DRAM traffic

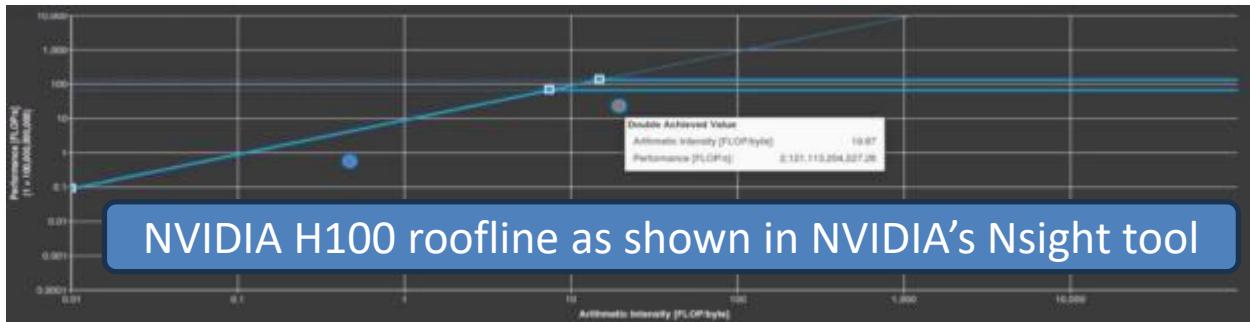


Roofline Model: Visual Performance Model

- Bound and bottleneck analysis (like Amdahl's law)
- Relates processor performance to off-chip memory traffic (bandwidth often the bottleneck)



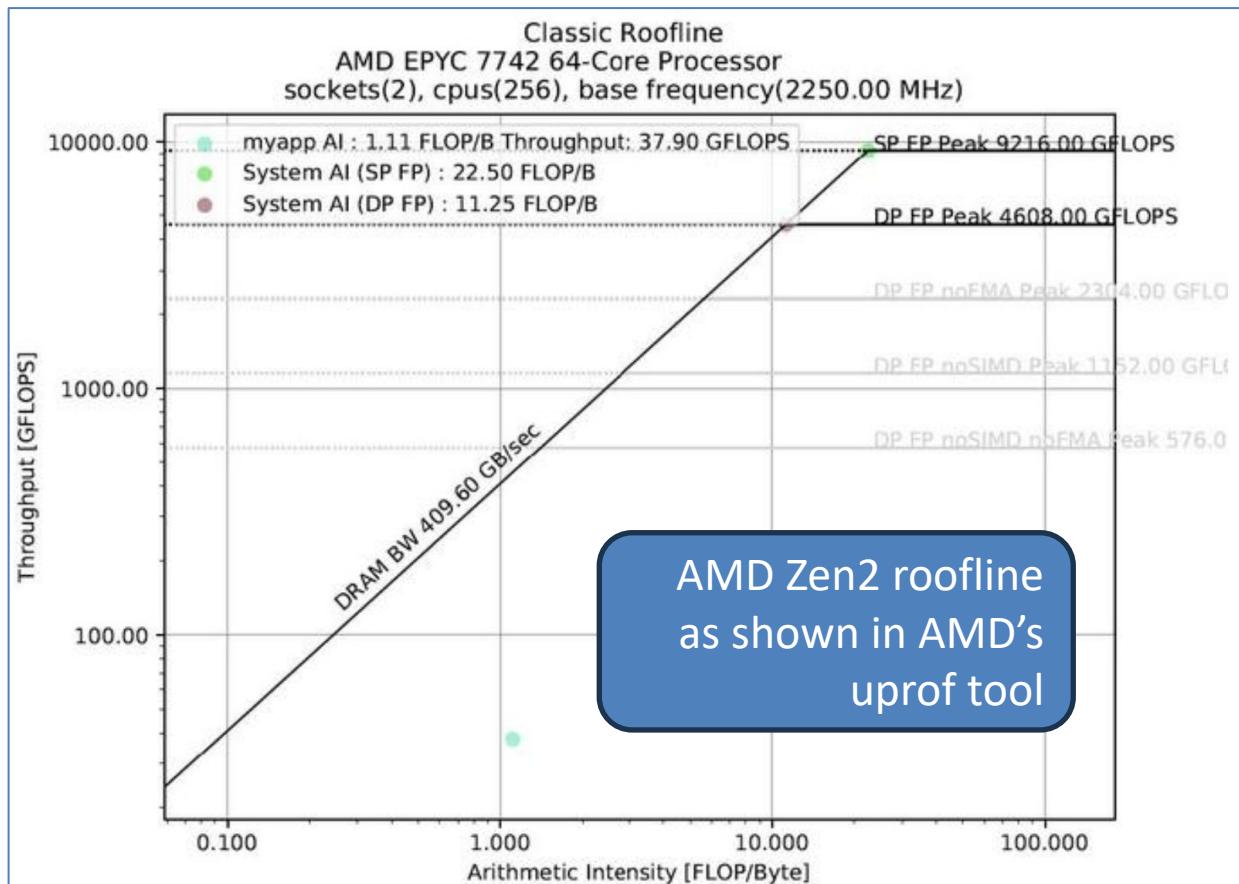
Roofline Model: Visual Performance Model



- The “roofline” concept is often used in performance optimisation tools
- To show how close you are to the limit
- And what you can do about it

<https://developer.nvidia.com/blog/accelerating-hpc-applications-with-nsight-compute-roofline-analysis/>

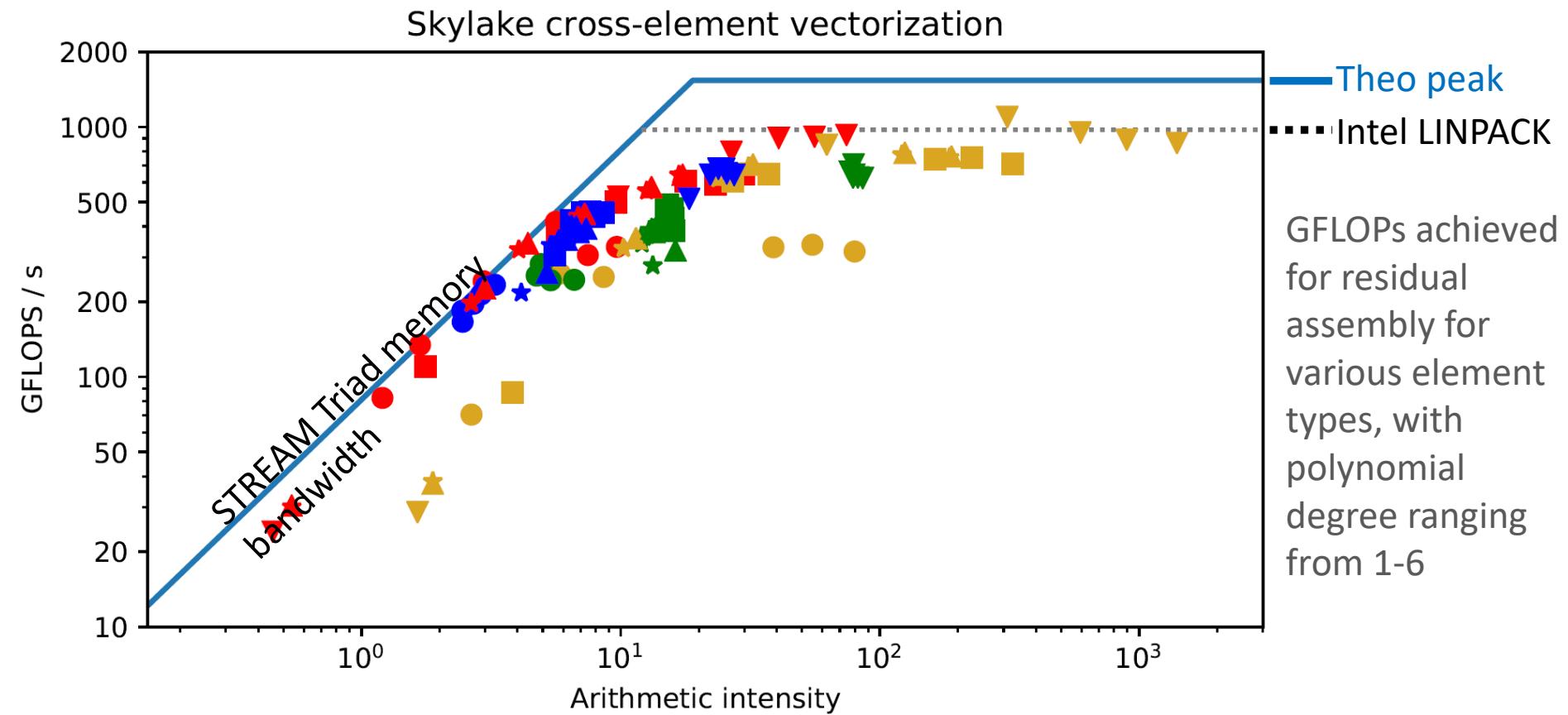
- The ridge point offers insight into the computer's overall performance potential
- It tells you whether your application *should be* limited by memory bandwidth, or by arithmetic capability



<https://docs.amd.com/r/en-US/57368-uProf-user-guide/Command-Line-Options?tocId=tGwsnZYhKNayV9CCIRa18Q>

Example from my research: Firedrake: single-node AVX512 performance

6



Firedrake implements a domain-specific language for partial differential equations – different equations, and different discretisations – have differing arithmetic intensity:

● mass - tri	■ helmholtz - tri	★ laplacian - tri	▲ elasticity - tri	▼ hyperelasticity - tri
● mass - quad	■ helmholtz - quad	★ laplacian - quad	▲ elasticity - quad	▼ hyperelasticity - quad
● mass - tet	■ helmholtz - tet	★ laplacian - tet	▲ elasticity - tet	▼ hyperelasticity - tet
● mass - hex	■ helmholtz - hex	★ laplacian - hex	▲ elasticity - hex	▼ hyperelasticity - hex

[Skylake Xeon Gold 6130 (on all 16 cores, 2.1GHz, turboboost off, Stream: 36.6GB/s, GCC7.3 –march=native)]

A study of vectorization for matrix-free finite element methods, Tianjiao Sun et al

<https://arxiv.org/abs/1903.08243>

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”
- Predicate registers k0-k7 (k0 is always true)
 - Each register holds a predicate *per operand* (per “lane”)
 - So each k register holds (up to) 64 bits*
- Rich set of instructions operate on 512-bit operands

* k registers are 64 bits in the AVX512BW extension; the default is 16

AVX512: vector addition

- Assembler:
 - `VADDPS zmm1 {k1}{z}, zmm2, zmm3`
- In C the compiler provides “vector intrinsics” that enable you to emit specific vector instructions, eg:
 - `res = _mm512_maskz_add_ps(k, a, b);`
- Only lanes with their corresponding bit set in predicate register `k1` (`k` above) are activated
- Two predication modes: *masking* and *zero-masking*
 - With “zero masking” (shown above), inactive lanes produce zero
 - With “masking” (omit “z” or “{z}”), inactive lanes do not overwrite their prior register contents

AVX12: vector addition

- Assembler:
 - VADDPS zmm1 {k1}{z}, zmm2, zmm3
- In C the compiler provides “vector intrinsics” that enable you to emit specific vector instructions, eg:
 - `res = _mm512_maskz_add_ps(k, a, b);`
- Only lanes with their corresponding bit in k1 are activated
- Two predication modes: *masking* and *zero-masking*
 - With “zero masking” (shown above), inactive lanes produce zero
 - With “masking” (omit “z” or “{z}”), inactive lanes do not overwrite their prior register contents

More formally...

FOR $j \leftarrow 0$ TO $KL-1$

$i \leftarrow j * 32$

IF $k1[j]$ OR *no writemask*

THEN $DEST[i+31:i] \leftarrow SRC1[i+31:i] + SRC2[i+31:i]$

ELSE

IF *merging-masking* ; merging-masking

THEN * $DEST[i+31:i]$ remains unchanged*

ELSE ; zeroing-masking

$DEST[i+31:i] \leftarrow 0$

FI

FI;

ENDFOR;

Can we get the compiler to vectorise?

Secure | <https://godbolt.org>

Compiler Explorer Editor Diff View More Share Other

C++ source #1 x

A Save/Load + Add new... C++ ▾

```

1 float c[1024];
2 float a[1024];
3 float b[1024];
4 void add ()
5 {
6     for (int i=0; i < 1024; i++)
7         c[i]=a[i]+b[i];
8 }
```

x86-64 gcc 5.4 (Editor #1, Compiler #1) C++ x

x86-64 gcc 5.4 -O3 -fopt-info

A 11010 .LX0: .text // \s+ Intel Demangle

Libraries + Add new...

```

1 _Z3addv:
2     xorl    %eax, %eax
3 .L2:
4     movaps  a(%rax), %xmm0
5     addq    $16, %rax
6     addps   b-16(%rax), %xmm0
7     movaps  %xmm0, c-16(%rax)
8     cmpq    $4096, %rax
9     jne     .L2
10    rep ret
11 b:
12    .zero   4096
13 a:
14    .zero   4096
15 c:
16    .zero   4096
```

Output (0/1) g++ (GCC-Explorer-Build) 5.4.0 - cached (4432)

In sufficiently simple cases, no problem:

Gcc reports:

test.c:6:3: note: loop vectorized

Secure | <https://godbolt.org>

Compiler Explorer Editor Diff View More Share Other

C++ source #1 x

A Save/Load + Add new... C++ .

```

1 float c[1024];
2 float a[1024];
3 float b[1024];
4 void add (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++ x

x86-64 gcc 5.4 -O3 -fopt-info

A 11010 .LX0: .text // \s+ Intel Demangle Libraries + Add new...

```

1 .L3addi:
2     testl %edi, %edi
3     jle .L1
4     leal -4(%rdi), %edx
5     leal -1(%rdi), %ecx
6     shr1 $2, %edx
7     addl $1, %edx
8     cmp1 $2, %ecx
9     leal 0(%rdx,4), %eax
10    jbe .L9
11    xorl %ecx, %ecx
12    xorl %esi, %esi
13 .L5:
14    movaps a(%rcx), %xmm0
15    addl $1, %esi
16    addq $16, %rcx
17    addps b-16(%rcx), %xmm0
18    movaps %xmm0, c-16(%rcx)
19    cmp1 %esi, %edx
20    ja .L5
21    cmp1 %edi, %eax
22    je .L12
23 .L3:
24    movslq %eax, %rdx
25    movss b(%rdx,4), %xmm0
26    addss a(%rdx,4), %xmm0
27    movss %xmm0, c(%rdx,4)
28    leal 1(%rax), %edx
29    cmp1 %edx, %edi
30    jle .L1
31    movslq %edx, %rdx
32    addl $2, %eax
33    movss a(%rdx,4), %xmm0
34    cmp1 %eax, %edi
35    addss b(%rdx,4), %xmm0
36    movss %xmm0, c(%rdx,4)
37    jle .L1
38    ctq
39    movss a(%rax,4), %xmm0
40    addss b(%rax,4), %xmm0
41    movss %xmm0, c(%rax,4)
42    ret
43 .L1:
44     rep ret
45 .L12:
46     rep ret
47 .L9:
48     xorl %eax, %eax
49     jmp .L3
50 b:
51     .zero 4096
52 a:
53     .zero 4096
54 c:
55     .zero 4096
```

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

Output (0/3) g++ (GCC-Explorer-Build) 5.4.0 - 377ms (5760B)

```
C++ source #1 x
A+ Save/Load + Add new... C++ x
1 void add(float * __restrict__ c,
2         float * __restrict__ a,
3         float * __restrict__ 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++ x
x86-64 gcc 5.4 -O3 -fopt-info
A+ 11010 LX0: .text // ls+ Intel Demangle Libraries + Add new...
1 .LaddPS_5_ll:
2     testl    Neax, Neax
3     pushl    Nr13
4     pushl    Nr12
5     pushl    Nr10
6     pushl    Nr9x
7     jne     .L1
8     movl    Nr15, Neax
9     leal    10(Nr0), Nr9d
10    andl   $15, Neax
11    shr    $2, Neax
12    negl    Neax
13    andl   $3, Neax
14    cmpl   Nr15, Neax
15    cmovl  Nr15, Neax
16    cmpl   $4, Nr9d
17    ja     .L23
18    movl    Nr16, Neax
19 .L3:
20    movss  (Nr13), Nmm0
21    cmpl   $1, Neax
22    movl    $1, Nr9d
23    addss  (Nr0), Nmm0
24    movss  Nmm0, (Nr0d)
25    je     .L5
26    movss  (Nr11), Nmm0
27    cmpl   $1, Neax
28    movl    $1, Nr9d
29    addss  (Nr0), Nmm0
30    movss  Nmm0, 4(Nr0d)
31    je     .L5
32    movss  8(Nr11), Nmm0
33    cmpl   $1, Neax
34    movl    $1, Nr9d
35    addss  8(Nr0), Nmm0
36    movss  Nmm0, 8(Nr0d)
37    je     .L5
38    movss  12(Nr11), Nmm0
39    cmpl   $4, Neax
40    addss  12(Nr0), Nmm0
41    movss  Nmm0, 12(Nr0d)
42 .L5:
43    cmpl   Neax, Nr9d
44    je     .L1
45 .L4:
46    subl    Neax, Nr9d
47    movl    Neax, Nebx
48    movl    Neax, Nr11d
49    leal    -4(Nr9), Nebx10d
50    subl    Neax, Nebx
51    shr    $2, Nr10d
52    addl   $1, Nr10d
53    cmpl   $2, Neax
54    leal    0(Nr11d), Nebx
55    je     .L7
56    leaq  0(Nr11d), Neax
57    xorl    Neax, Neax
58    leaq  (Nr11, Neax), Nr13
59    leaq  (Nr0, Neax), Nr12
60    leaq  (Nr11, Neax), Nr11
61    xorl    Neax, Neax
62 .L9:
63    movss  (Nr12, Neax), Nmm0
64    addss  8(Nr11, Neax), Nmm0
65    addss  12(Nr11, Neax), Nmm0
66    movss  Nmm0, (Nr11, Neax)
67    addss  $16, Neax
68    cmpl   Neax, Nr11d
69    ja     .L8
70    addl   $1, Nr11d
71    cmpl   Neax, Nr9d
72    je     .L1
73 .L7:
74    movss  Nr13, Neax
75    movss  (Nr11, Neax, 4), Nmm0
76    addss  8(Nr11, Neax, 4), Nmm0
77    movss  Nmm0, (Nr11, Neax, 4)
78    leal    10(Nr9), Neax
79    cmpl   Neax, Neax
80    jle     .L1
81    cmovl  Nr13, Neax
82    addl   $2, Nr9d
83    movss  (Nr11, Neax, 4), Nmm0
84    cmpl   Nr13, Neax
85    addss  (Nr0, Neax, 4), Nmm0
86    movss  Nmm0, (Nr0, Neax, 4)
87    jle     .L1
88    movl    Nr12, Nebx
89    movss  (Nr11, Neax, 4), Nebx
90    addss  (Nr0, Neax, 4), Nebx
91    movss  Nmm0, (Nr0, Neax, 4)
92 .L11:
93    popl    Nr13
94    popl    Nr12
95    popl    Nr11
96    popl    Nr10
97    ret
98 .L23:
99    testl    Neax, Neax
100   jne     .L14
101   xorl    Nr16, Nr9d
102   smp     .L4
```

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

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
```

I asked chatgpt to turn this back into C code:

```
#include <immintrin.h> // For SSE/AVX instructions

void process_arrays(float *dst, const float *src1, const float *src2, int n) {
    if (n <= 0) {
        return; // No work to do for non-positive sizes
    }

    int aligned_start = ((uintptr_t)src1 & 15) >> 2; // Align offset calculation
    aligned_start = (-aligned_start) & 3; // Calculate start for alignment
    if (aligned_start > n) {
        aligned_start = n; // Prevent overflow
    }

    int remainder = n > 4 ? aligned_start : n;

    // Process initial unaligned elements
    for (int i = 0; i < remainder; ++i) {
        dst[i] = src1[i] + src2[i];
    }

    int processed = remainder;

    // Vectorized processing
    if (n > 4) {
        int chunks = (n - remainder) / 4;
        for (int i = 0; i < chunks; ++i) {
            __m128 vec1 = _mm_loadu_ps(src1 + processed); // Load 4 floats from src1
            __m128 vec2 = _mm_loadu_ps(src2 + processed); // Load 4 floats from src2
            __m128 result = _mm_add_ps(vec1, vec2); // Add the vectors
            _mm_storeu_ps(dst + processed, result); // Store result in dst
            processed += 4;
        }
    }

    // Process remaining elements
    while (processed < n) {
        dst[processed] = src1[processed] + src2[processed];
        ++processed;
    }
}
```

```
C++ source #1 x
A Save/Load + Add new... C++ x
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++ x -O3 -fopt-info

1 _Faddf2_5_1:
2 testl %rcx, %rcx
3 jne .L27
4 leal 16(%rdi), %rcx
5 leal 16(%rsi), %rdi
6 leal 16(%rdx), %rdx
7 cmpb %rcx, %rdi
8 setbo %rdx
9 cmpl %rdx, %rdi
10 setbo %rdx
11 orl %rdx, %rdx
12 leal 16(%rdx), %rdx
13 cmpl %rdx, %rdx
14 setbo %rdx
15 cmpl %rdx, %rdx
16 setbo %rdx
17 orl %rdx, %rdx
18 testb %rdx, %rdx
19 jne .L3
20 cmpl %rdx, %rdx
21 jne .L3
22 movq %rdx, %rdx
23 pushq %rdx
24 pushq %rdx
25 andl \$15, %rcx
26 pushq %rcx
27 pushq %rcx
28 shrl \$2, %rcx
29 negl %rcx
30 andl \$15, %rcx
31 cmpl %rdx, %rdx
32 cmovl %rdx, %rdx
33 xorl %rdx, %rdx
34 testl %rdx, %rdx
35 jne .L4
36 movss (%rdi,%rdx), %xmm0
37 cmpl \$1, %xmm0
38 movl %xmm0, (%rdi,%rdx)
39 addss %xmm0, %xmm0
40 movss %xmm0, (%rdi,%rdx)
41 je .L5
42 movss (%rdi,%rdx), %xmm0
43 cmpl \$1, %xmm0
44 movl %xmm0, (%rdi,%rdx)
45 addss %xmm0, %xmm0
46 movss %xmm0, (%rdi,%rdx)
47 je .L5
48 movss (%rdi,%rdx), %xmm0
49 movl %xmm0, (%rdi,%rdx)
50 addss %xmm0, %xmm0
51 movss %xmm0, (%rdi,%rdx)
52 .L4:
53 subl %rcx, %rdx
54 salq %rdx, %rdx
55 udivl %rdx, %rdx
56 leal -4(%rdx), %rdi
57 leal (%rdi,%rdx), %rdx
58 leal (%rdx,%rdx), %rdx
59 udivl %rdx, %rdx
60 addq %rdx, %rdx
61 shrq %rdx, %rdx
62 addq %rdx, %rdx
63 leal 8(%rdx,%rdx), %rdx
64 .L5:
65 movups (%rdx,%rdx), %xmm0
66 addl \$1, %rdx
67 addss %xmm0, (%rdx,%rdx)
68 movss %xmm0, (%rdx,%rdx)
69 addq %rdx, %rdx
70 cmpl %rdx, %rdx
71 jne .L7
72 movl %rdx, %rdx
73 cmpl %rdx, %rdx
74 jne .L11
75 movss (%rdx,%rdx), %xmm0
76 addss %xmm0, %xmm0
77 addss %xmm0, %xmm0
78 movss %xmm0, (%rdi,%rdx,%rdx)
79 leal 16(%rdx), %rcx
80 cmpl %rcx, %rcx
81 jne .L11
82 .L7:
83 addl \$1, %rdx
84 movss (%rdx,%rdx), %xmm0
85 cmpl %rdx, %rdx
86 addss %xmm0, %xmm0
87 movss %xmm0, (%rdi,%rdx,%rdx)
88 jne .L11
89 movl %rdx, %rdx
90 movss (%rdx,%rdx), %xmm0
91 addss %xmm0, %xmm0
92 movss %xmm0, (%rdi,%rdx,%rdx)
93 .L11:
94 popq %rdx
95 popq %rdx
96 popq %rdx
97 .L12:
98 rep ret
99
100 xorl %rcx, %rcx
101 .L12:
102 movss (%rdi,%rdx,%rdx), %xmm0
103 addss %xmm0, %xmm0
104 movss %xmm0, (%rdi,%rdx,%rdx)
105 addq %rdx, %rdx
106 cmpl %rdx, %rdx
107 jne .L12
108 rep ret

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
```

What to do if the compiler just won't vectorise your loop? Option #1: **ivdep pragma**

```
void add (float *c, float *a, float *b)
{
    #pragma ivdep
    for (int i=0; i <= N; i++)
        c[i]=a[i]+b[i];
}
```

IVDEP (Ignore Vector DEPendencies) compiler hint.

Tells compiler “Assume there are no loop-carried dependencies”

This tells the compiler vectorisation is *safe*: it might still not vectorise

What to do if the compiler just won't vectorise your loop? Option #2: OpenMP 4.0 pragmas

```
void add (float *c, float *a, float *b)
{
    #pragma omp simd
    for (int i=0; i <= N; i++)
        c[i]=a[i]+b[i];
}
```

Indicates that the loop can be transformed into a SIMD loop
(i.e. the loop can be executed concurrently using SIMD instructions)

```
#pragma omp declare simd
void add (float *c, float *a, float *b)
{
    *c=*a+*b;
}
```

"declare simd" can be applied to a function to enable SIMD instructions at the function level from a SIMD loop

Tells compiler “vectorise this code”. It might still not do it...

What to do if the compiler just won't vectorise your loop? Option #3: SIMD intrinsics:

```
void add (float *c, float *a, float *b)
{
    __m128* pSrc1 = (__m128*) a;
    __m128* pSrc2 = (__m128*) b;
    __m128* pDest = (__m128*) c;
    for (int i=0; i <= N/4; i++)
        *pDest++ = _mm_add_ps (*pSrc1++, *pSrc2++);
}
```

Vector instruction lengths are hardcoded in the data types and intrinsics

This tells the compiler which specific vector instructions to generate. This time it really will vectorise!

What to do if the compiler just won't vectorise your loop? Option #4: SIMD

Basically... think of each lane as a thread

Or: vectorise an *outer* loop:

```
#pragma omp simd
for (int i=0; i<N; ++i) {
    if (...) { ... } else { ... }
    for (int j=....) { ... }
    while (...) { ... }
    f (...)
```

In the body of the vectorised loop, each lane executes a different iteration of the loop – *whatever* the loop body code does

Use predication to handle:

- nested if-then-else
- While loops
- For loops
- Function calls

More later – when we look at GPUs



Add...

More

Watch C++ Weekly to learn new C++ features

Share

Other

Policies

C source #1

A Save/Load + Add new... Vim

C

```
1 // icc: -xCORE-AVX512 -qopt-zmm-usage=high -qopt
2 #define ALIGN __attribute__ ((aligned (64)))
3 //#define ALIGN
4
5 float ALIGN c[1024];
6 float ALIGN a[1024];
7 float ALIGN b[1024];
8
9
10 void add ()
11 {
12     for (int i=0; i < 1024; i++)
13         c[i]=a[i]+b[i];
14 }
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C

x86-64 icc 19.0.1

x

x

-xCORE-AVX512 -qopt-zmm-usage=high

A

 11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle

Libraries + Add new... Add tool...

```
1 add:
2     xor    eax, eax
3     ..B1.2:                      # Preds ..B1.2 ..B1.1
4     vmovups zmm0, ZMMWORD PTR [a+rax*4]
5     vaddps zmm1, zmm0, ZMMWORD PTR [b+rax*4]
6     vmovups ZMMWORD PTR [c+rax*4], zmm1
7     add    rax, 16
8     cmp    rax, 1024
9     jb    ..B1.2      # Prob 99%
10    vzeroupper
11    ret
```



Output (0/0) x86-64 icc 19.0.1 - 679ms (8614B)

#1 with x86-64 icc 19.0.1

x

A Wrap lines

Compiler returned: 0

godbolt.org

Watch C++ Weekly to learn new C++ features

C source #1 x

A Save/Load + Add new... Vim C

```
1 //icc: -xCORE-AVX512 -qopt-zmm-usage=high -qopt
2 #define ALIGN __attribute__ ((aligned (64)))
3 //#define ALIGN
4
5 float ALIGN c[1024];
6 float ALIGN a[1024];
7 float ALIGN b[1024];
8 int ALIGN ind[1024];
9
10 void add ()
11 {
12     for (int i=0; i < 1024; i++)
13         c[i]=a[i]+b[ind[i]];
14 }
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C x

x86-64 icc 19.0.1 -xCORE-AVX512 -qopt-zmm-usage=high

A

11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle

Libraries + Add new... Add tool...

```
1 add:
2     xor    eax, eax
3 ..B1.2:                                # Preds ..B1.2 ..B1.1
4     vmovups zmm0, ZMMWORD PTR [ind+rax*4]
5     vpcmpeqb k1, xmm0, xmm0
6     vpxord  zmm1, zmm1, zmm1
7     vgatherdps zmm1{k1}, DWORD PTR [b+zmm0*4]
8     vaddps  zmm2, zmm1, ZMMWORD PTR [a+rax*4]
9     vmovups ZMMWORD PTR [c+rax*4], zmm2
add    rax, 16
cmp   rax, 1024
jb    ..B1.2      # Prob 99%
vzeroupper
ret
```

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??)

#1 with x86-64 icc 19.0.1 x

A Wrap lines

Compiler returned: 0

Output (0/0) x86-64 icc 19.0.1 - 946ms (93598)



Add...

More

Watch C++ Weekly to learn new C++ features

Share

Other

Policies

C source #1

A Save/Load + Add new... Vim

C

```
1 //icc: -xCORE-AVX512 -qopt-zmm-usage=high -qopt
2 #define ALIGN __attribute__((aligned(64)))
3 //#define ALIGN
4
5 float ALIGN c[1024];
6 float ALIGN a[1024];
7 float ALIGN b[1024];
8
9 void add ()
10 {
11     for (int i=0; i < 1024; i++)
12 //     if (a[i]!=0.0)
13     c[i]=a[i]+b[i];
14 }
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C

x86-64 icc 19.0.1

-xCORE-AVX512 -qopt-zmm-usage=h

A

 11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle

Libraries + Add new... Add tool...

```
1 add:
2     xor    eax, eax
3 ..B1.2:                      # Preds ..B1.2 ..B1.1
4     vmovups zmm0, ZMMWORD PTR [a+rax*4]
5     vaddps  zmm1, zmm0, ZMMWORD PTR [b+rax*4]
6     vmovups ZMMWORD PTR [c+rax*4], zmm1
7     add    rax, 16
8     cmp    rax, 1024
9     jb    ..B1.2      # Prob 99%
10    vzeroupper
11    ret
```

Output (0/0) x86-64 icc 19.0.1 - 1086ms (8614B)

#1 with x86-64 icc 19.0.1

A Wrap lines

Compiler returned: 0



Add... ▾

More ▾

Watch C++ Weekly to learn new C++ features

Share ▾

Other ▾

Policies ▾

C source #1

A ▾ Save/Load + Add new... ▾ Vim

C

```

1 //icc: -xCORE-AVX512 -qopt-zmm-usage=high -qopt
2 #define ALIGN __attribute__ ((aligned (64)))
3 //#define ALIGN
4
5 float ALIGN c[1024];
6 float ALIGN a[1024];
7 float ALIGN b[1024];
8
9 void add ()
10 {
11     for (int i=0; i < 1024; i++)
12         if (a[i]!=0.0)
13             c[i]=a[i]+b[i];
14 }
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C

x86-64 icc 19.0.1



-xCORE-AVX512 -qopt-zmm-usage=h

A ▾

 11010
 ./a.out
 .LX0:
 lib.f:
 .text
 // \s+
 Intel
 Demangle

Libraries ▾ + Add new... ▾ Add tool... ▾

```

1 add:
2     xor    eax, eax
3     vpxord    zmm0, zmm0, zmm0
4 ..B1.2:          # Preds ..B1.2 ..B1.1
5     vmovups   zmm1, ZMMWORD PTR [a+rax*4]
6     vcmpps    k1, zmm1, zmm0, 4
7     vaddps    zmm2, zmm1, ZMMWORD PTR [b+rax*4]
8     vmovups   ZMMWORD PTR [c+rax*4]{k1}, zmm2
9     add     rax, 16
10    cmp     rax, 1024
11    jb      ..B1.2          # Prob 99%
12    vzeroupper
13    ret
```

Conditional: a[i]!=0.0

We have a register containing a vector of Boolean predicates

We use a *predicated* vector instruction
Lanes with inactive predicates are idle

Output (0/0) x86-64 icc 19.0.1 - cached (8867B)

#1 with x86-64 icc 19.0.1

A ▾ Wrap lines

Compiler returned: 0

Vector execution alternatives

Implementation may execute n-wide vector operation with an n-wide ALU
– or maybe in smaller, m-wide blocks

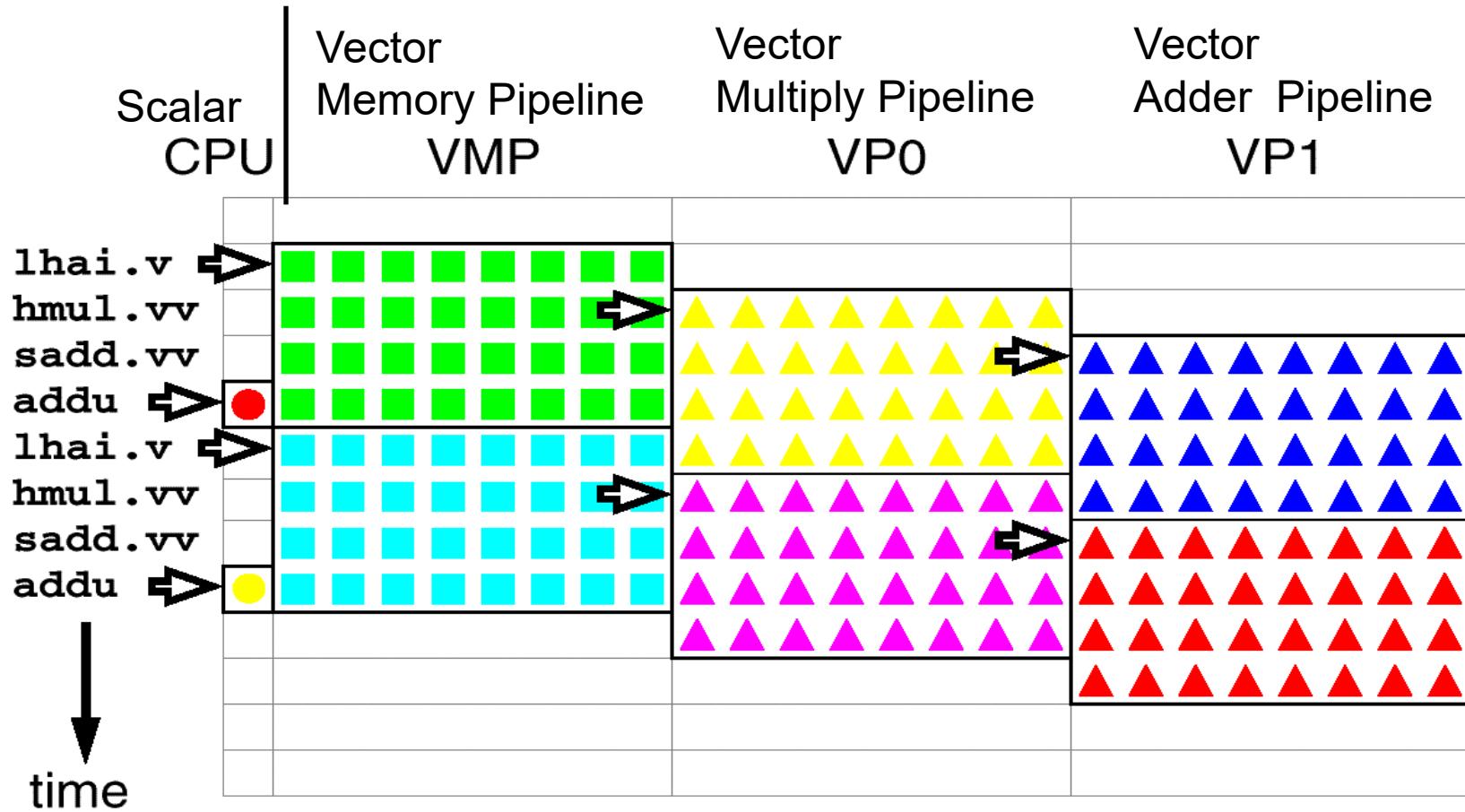
► **vector pipelining:**

- Consider a simple static pipeline
- Vector instructions are executed serially, element-by-element, using a pipelined FU – or in n-wide chunks if your FU is n-wide
- We have several pipelined FUs
- “vector chaining” – each word is forwarded to the next instruction as soon as it is available
- FUs form a long pipelined chain

► **uop decomposition:**

- Consider a dynamically-scheduled o-o-o machine
- Each n-wide vector instruction is split into m-wide uops at decode time
- The dynamic scheduling execution engine schedules their execution, possibly across multiple FUs
- They are committed together

Vector pipelining – “chaining”



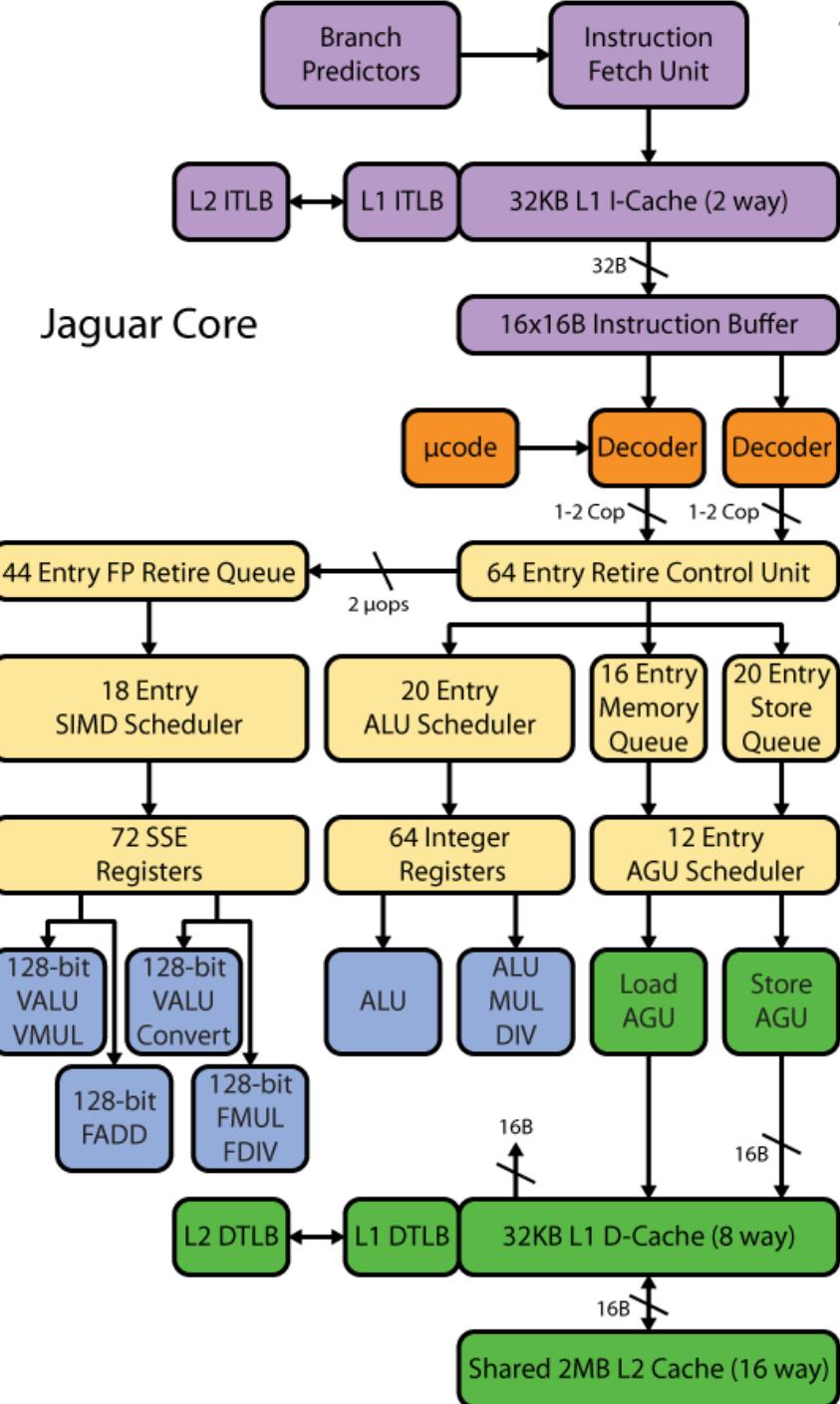
● ■ ▲ Operations
➡ Instruction issue

- Vector FUs are 8-wide - each 32-wide vector instruction is executed in 4 blocks
- Forwarding is implemented block-by-block
- So memory, mul, add and store are chained together into one continuously-active pipeline

Uop decomposition - example

AMD Jaguar

- **Low-power 2-issue dynamically-scheduled processor core**
- **Supports AVX-256 ISA**
- **Has two 128-bit vector ALUs**
- **256-bit AVX instructions are split into two 128-bit uops, which are scheduled independently**
- **Until retirement**
- **A “zero-bit” in the rename table marks a register which is known to be zero**
- **So no physical register is allocated and no redundant computation is done**



SIMD Architectures: discussion

- Reduced Turing Tax: more work, fewer instructions
- Relies on compiler or programmer
- Simple loops are fine, but many issues can make it hard
- “lane-by-lane” predication allows conditionals to be vectorised, but branch divergence may lead to poor utilisation
- Indirections can be vectorised on some machines (`vgather`, `vscatter`) but remain hard to implement efficiently unless accesses happen to fall on a small number of distinct cache lines
- Vector ISA allows broad spectrum of microarchitectural implementation choices
- Intel’s vector ISA has grown enormous as vector length has been successively increased
- ARM’s “scalable vector extension” (SVE) is an ISA design that hides the vector length (by using a special loop branch)

Topics we have not had time to cover

- **ARM's SVE, RISCV vector extensions:**
 - ▶ a vector ISA that achieves binary compatibility across machines with different vector width and uop decomposition
- **Matrix registers and matrix instructions**
 - ▶ Eg Nvidia's “tensor cores”
- **Exotic vector instructions**
 - ▶ Collision detect (how to vectorise, for example, histogramming)
 - ▶ Permutations
 - ▶ Complex arithmetic
- **Pipelined vector architectures:**
 - ▶ The classical vector supercomputer
- **Whole-function vectorisation, ISPC, SIMD**
 - ▶ Vectorising nested conditionals
 - ▶ Vectorising non-innermost loops
 - ▶ Vectorising loops containing while loops
- **SIMD and the relationship/similarities with GPUs**
 - ▶ Coming!

Vectors, units, lanes

another attempt to clear up confusion

- Let's consider Intel's AVX512 instruction set and its implementation on Skylake processors (all this applies to other ISAs more or less).
- AVX512 has 32 vector registers, each 512 bits long (called "zmm0"- "zmm31"). Each register can hold a vector - eg a vector of 16 32-bit floats (or 8 64-bit doubles). A vector add instruction does element-wise vector addition on two vector registers, yielding a third 512-bit result. A vector FMA ("fused multiply-add") does $r[0:15]+=a[0:15]*b[0:15]$ in one instruction.
- Some Skylake products have just one arithmetic unit for executing such instructions, but some fancy ones have two AVX512 vector execution units. The Skylake microarchitecture can issue up to about 4 instructions per cycle, so two out of every four instructions needs to be a vector FMA if you want to get maximum performance on such a machine.
- The word "lane" is used when you want to think about a sequence of vector instructions, but you want to focus on just one element at a time - a vertical slice through the instruction sequence.
- The word "lane" refers to the same idea as what is sometimes called "single-instruction, multiple thread" (SIMT). This is how GPUs are programmed - its the idea behind CUDA and OpenCL. Imagine a loop consisting of scalar (ie non-vector) instructions. That's the SIMT "view" of your code - you see what is happening "lanewise". Now expand every instruction in the loop into a vector instruction - so the loop does what it does on a vector of 16 lanes of data. This is the "SIMT->SIMD translation".
- SIMT to SIMD translation gets tricky if the loop body contains an if-then. For this, AVX512 uses the idea of "predication". For this purpose it has one-bit-per-lane predicate registers k0-k7. These registers can be used to control which lanes of a vector instruction are active and which lanes do nothing.

Summary Vectorisation Solutions

1. Indirectly through **high-level libraries/code generators**
2. **Auto-vectorisation** (eg use “-O3 –mavx2 –fopt-info” and hope it vectorises):
 - code complexity, sequential languages and practices get in the way
 - Give your **compiler hints** and hope it vectorises:
 - C99 "restrict" (implied in FORTRAN since 1956)
 - `#pragma ivdep`
3. **Code explicitly**:
 - In assembly language
 - SIMD instruction intrinsics
 - OpenMP 4.0 `#pragma omp simd`
 - Kernel functions:
 - OpenMP 4.0: `#pragma omp declare simd`
 - OpenCL or CUDA: more later

- Fun question if you like this sort of thing....
 - What is “vzeroupper” for?

```

1  add:
2      xor      eax, eax
3  ..B1.2:                      # Preds ..B1.2 ..B1.1
4      vmovups  zmm0, ZMMWORD PTR [a+rax*4]
5      vaddps   zmm1, zmm0, ZMMWORD PTR [b+rax*4]
6      vmovups  ZMMWORD PTR [c+rax*4], zmm1
7      add      rax, 16
8      cmp      rax, 1024
9      jb       ..B1.2          # Prob 99%
10     vzeroupper
11     ret

```

Compiler Explorer interface showing C++ source code and assembly output for x86-64 clang (trunk).

C++ Source:

```

1 #include <string.h>
2
3 void f(char* a, char* b) {
4     memcpy(a, b, 32);
5 }

```

Assembly Output:

```

x86-64 clang (trunk) (Editor #1, Compiler #1) C++ -O3 -mavx
f(char*, char*):
    vmovups ymm0, ymmword ptr [rsi]
    vmovups ymmword ptr [rdi], ymm0
    vzeroupper
    ret

```

The assembly output shows the generated assembly code for the `f` function, including the `vzeroupper` instruction at the end.

More things in case you're interested...

https://godbolt.org

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

COMPILER EXPLORER Add... More Templates

C++ source #1

A + v s 🔍 ✖

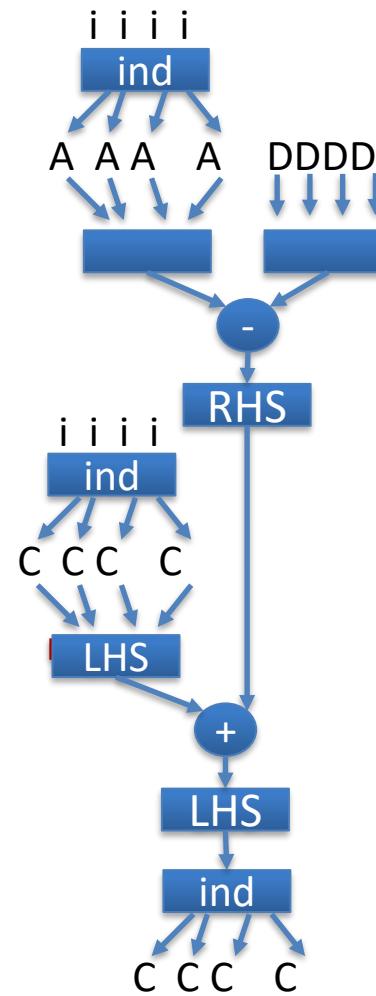
C++

```

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. $\text{RHS}[0:16] = \text{A}[ind[i:i+16]] - \text{D}[i:i+16]$
5. Gather the $\text{LHS}[0:16] = \text{C}[ind[i:i+16]]$
6. Add (`+=`): $\text{LHS}[0:16] += \text{RHS}[0:16]$
7. Scatter: $\text{C}[ind[i:i+16]] = \text{LHS}[0:16]$



https://godbolt.org

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

COMPILER EXPLORER Add... More Templates

C++ source #1

A C++

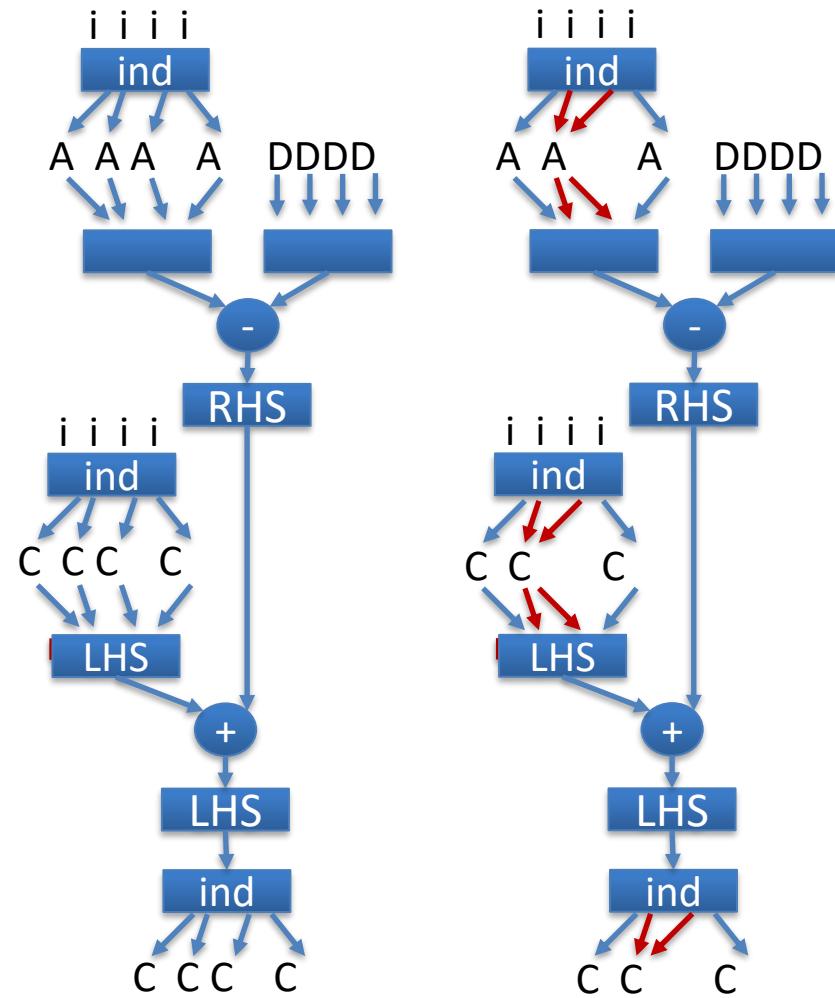
```

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. $\text{RHS}[0:16] = \text{A}[ind[i:i+16]] - \text{D}[i:i+16]$
5. Gather the $\text{LHS}[0:16] = \text{C}[ind[i:i+16]]$
6. Add ($+=$): $\text{LHS}[0:16] += \text{RHS}[0:16]$
7. Scatter: $\text{C}[ind[i:i+16]] = \text{LHS}[0:16]$

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

A C++

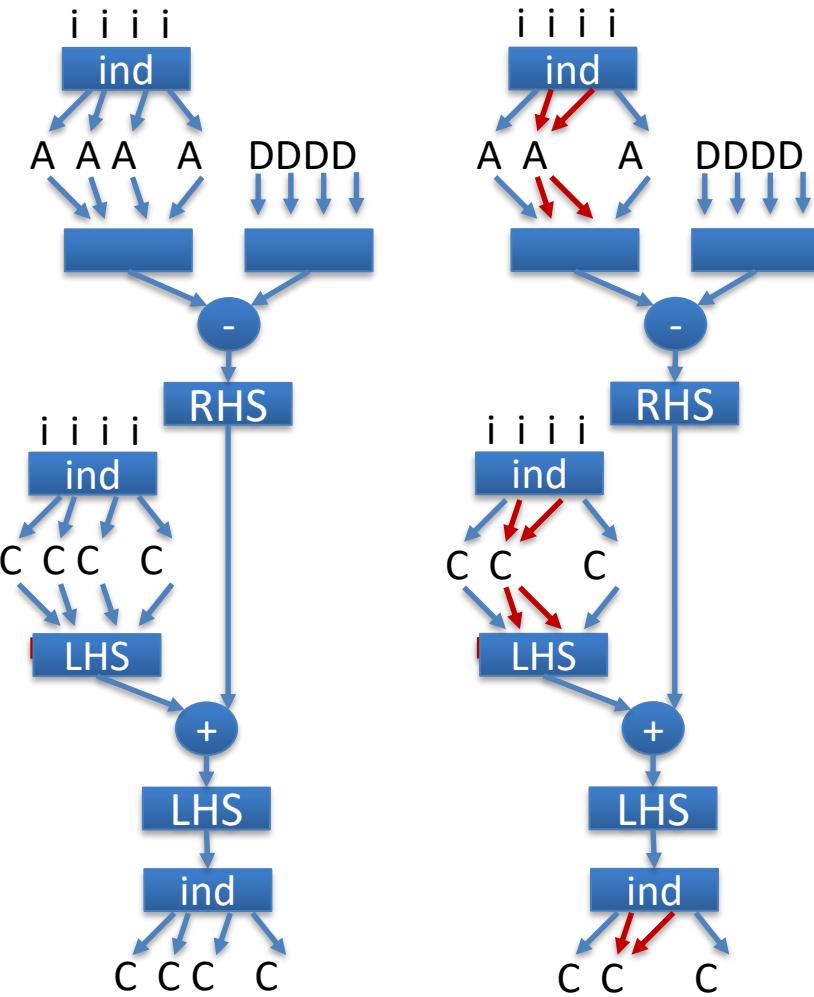
```

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: $\text{ind}[i]$

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

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 Engine Home - BBC News Staff travel and expense Shareable Whiteboard The

COMPILER EXPLORER

Add... More Templates

C++ source #1

A 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 $\text{ind}[i:i+16]$
2. Gather a vector $\text{A}[\text{ind}[i:i+16]]$
3. Subtract the $\text{D}[i]$ values:
4. $\text{RHS}[0:16] = \text{A}[\text{ind}[i:i+16]] - \text{D}[i:i+16]$
5. Gather the $\text{LHS}[0:16] = \text{C}[\text{ind}[i:i+16]]$
6. Add ($+=$): $\text{LHS}[0:16] += \text{RHS}[0:16]$
7. Scatter: $\text{C}[\text{ind}[i:i+16]] = \text{LHS}[0:16]$

3 .LCPI0_1:
4 .long 63
5 P():
6 vpbroadcastd ymm0, dword ptr [rip + .LCPI0_0]
7 vpbroadcastd ymm1, dword ptr [rip + .LCPI0_1]
8 vcpmeqd ymm2, ymm2, ymm2
9 xor eax, eax
10 jmp .LBB0_1
11 .LBB0_7:
12 vcpmeqb k1, xmm0, xmm0
13 vpxor xmm6, xmm6, xmm6
14 vpmovqd ymm3, zmm3
15 Add RHS into LHS
16 lea rax, [rax + 8]
17 add rax, 16
18 vgatherdd ymm6 {k1}, ymmword ptr [4*ymm4 + rax]
19 vcpmeqb k1, xmm0, xmm0
20 vpadd ymm4, ymm6, ymm5
21 vpscatterdd ymmword ptr [4*ymm3 + rax] {k1}, ymm4
22 cmp rax, 10232
23 jae .LBB0_8
24 .LBB0_1:
25 vmovdqu ymm4, ymmword ptr [4*rax + ind]
26 vcpmeqb k1, xmm0, xmm0
27 vpxor xmm5, xmm5, xmm5
28 vpmovsdq ymm5, zmm3
29 vgatherrd ymm5 {k1}, ymmword ptr [4*ymm4 + A]
30 vpsubd ymm5, ymm5, ymmword ptr [4*rax + D]
31 vcpconflictq zmm6, zmm6
32 vpmovsdq ymm6, zmm6
33 vpxor ymm7, ymm6, ymm0
34 If no conflicts the
35 skip to fast case
36 vptest ymm7, ymm7
37 je .LBB0_4
38 .LBB0_3:
39 vpermrd ymm7, ymm6, ymm5
40 vpermrd ymm6 {k1}, ymm6, ymm6
41 vpadd ymm5 {k1}, ymm5, ymm7
42 vcpmpneqd k1, ymm6, ymm2
43 vptest ymm6, ymm2
44 jae .LBB0_3
45 .LBB0_4:
46 vcpmeqb k1, xmm0, xmm0
47 vpxor xmm6, xmm6, xmm6
48 vpmovqd ymm3, zmm3
49 vgatherrd ymm6 {k1}, ymmword ptr [4*ymm4 + C]
50 vcpmeqb k1, xmm0, xmm0
51 vpadd ymm4, ymm0, ymm5
52 vpxor xmm5, xmm5, xmm5
53 vpscatterd ymmword ptr [4*ymm3 + C] {k1}, ymm4
54 vcpmeqb k1, xmm0, xmm0
55 vmovdqu ymm4, ymmword ptr [4*rax + ind+32]
56 vpmovsdq ymm5, zmm3
57 vgatherrd ymm5 {k1}, ymmword ptr [4*ymm4 + A]
58 vpsubd ymm5, ymm5, ymmword ptr [4*rax + D+32]
59 vcpconflictq zmm6, zmm6
60 vpmovsdq ymm6, zmm6
61 vpxor ymm7, ymm6, ymm0
62 vptest ymm7, ymm7
63 je .LBB0_7
64 vcpmpneqd k1, ymm6, ymm0
65 vpsubd ymm6, ymm1, ymm6
66 .LBB0_6:
67 vpermrd ymm7, ymm6, ymm5
68 vpermrd ymm6 {k1}, ymm6, ymm6
69 vpadd ymm5 {k1}, ymm5, ymm7
70 vcpmpneqd k1, ymm6, ymm2
71 vptest ymm6, ymm2
72 jae .LBB0_6
73 jmp .LBB0_7
74 .LBB0_8:
75 vzeroupper
76 ret

vpconflictq
instruction checks
for duplicate values
in $\text{ind}[i:i+16]$

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

Roughly...

Unrolled copy

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

Not examinable

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

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: algorithms for parallelising compilers - 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

Example 4:

```
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])
```

```
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*i-10] + 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!

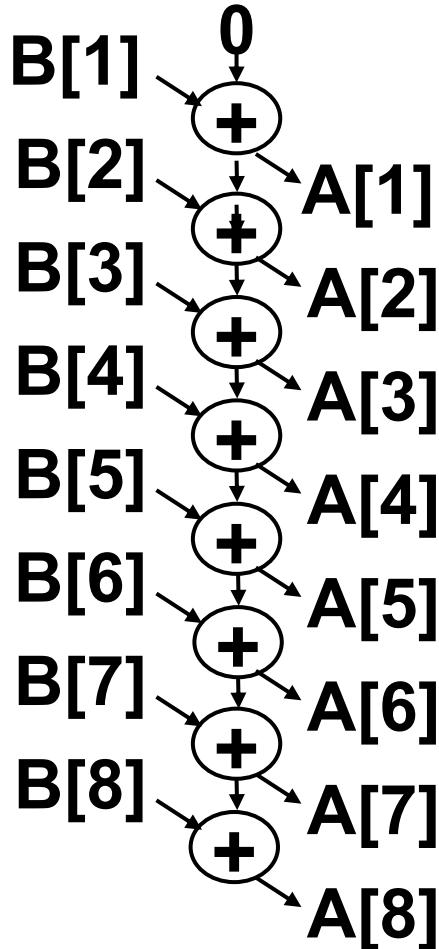
S1 : A[0] := 0

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



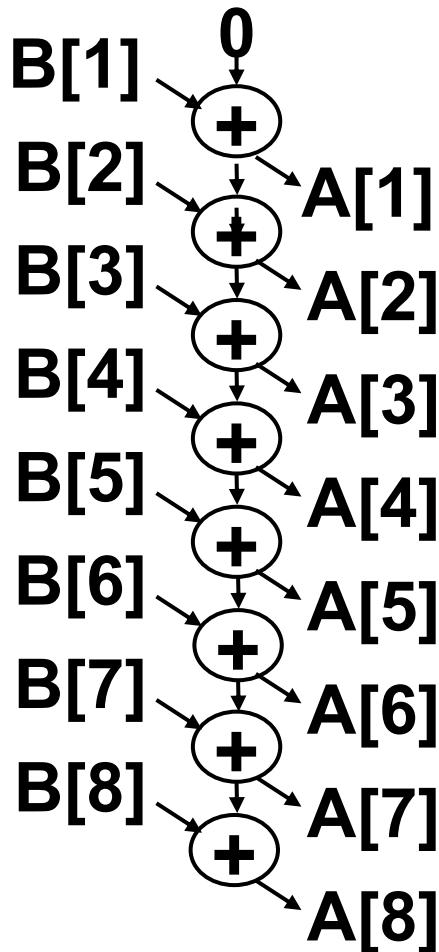
S1 : A[0] := 0

for i = 1 to 8

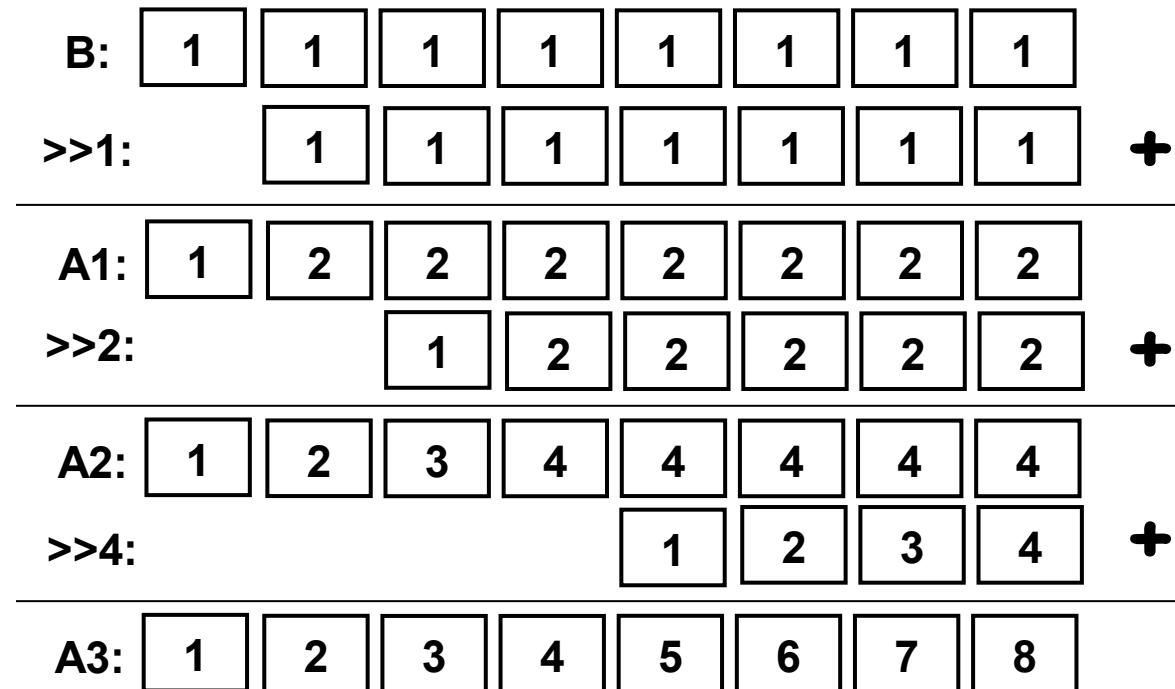
S2 : A[i] := A[i-1] + B[i]

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

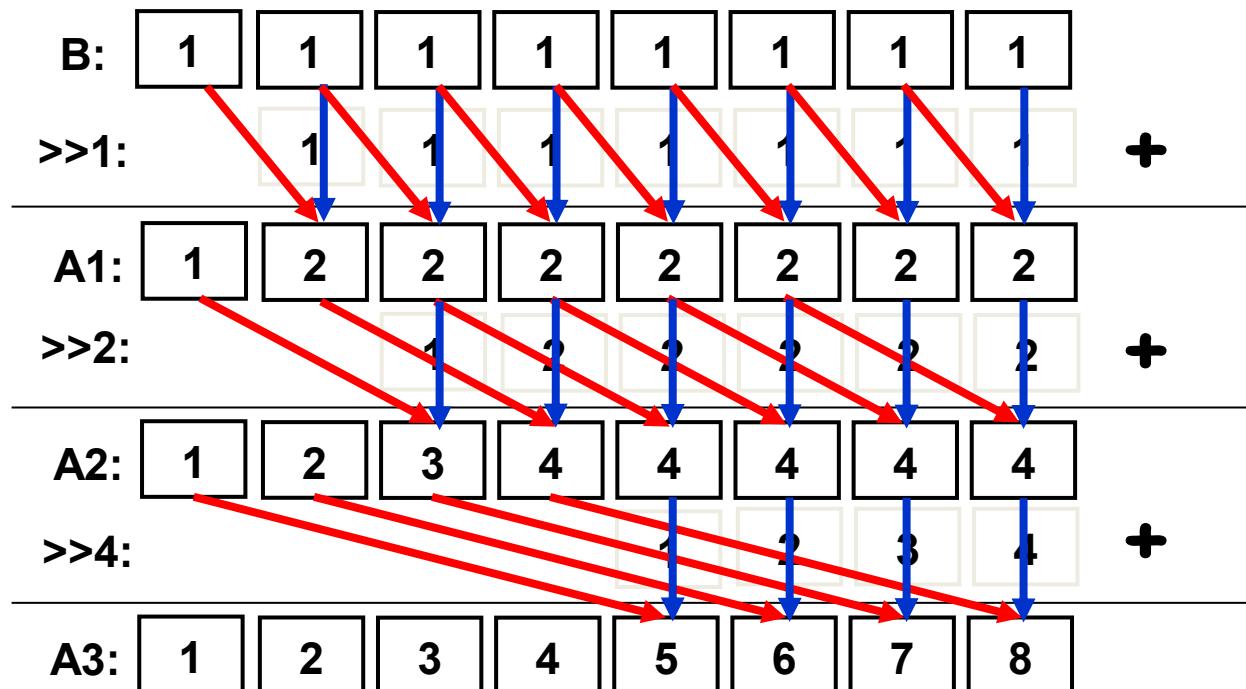
for i = 1 to 8

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



“Parallel scan” or “parallel prefix sum”

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

S1 : A[0] := 0

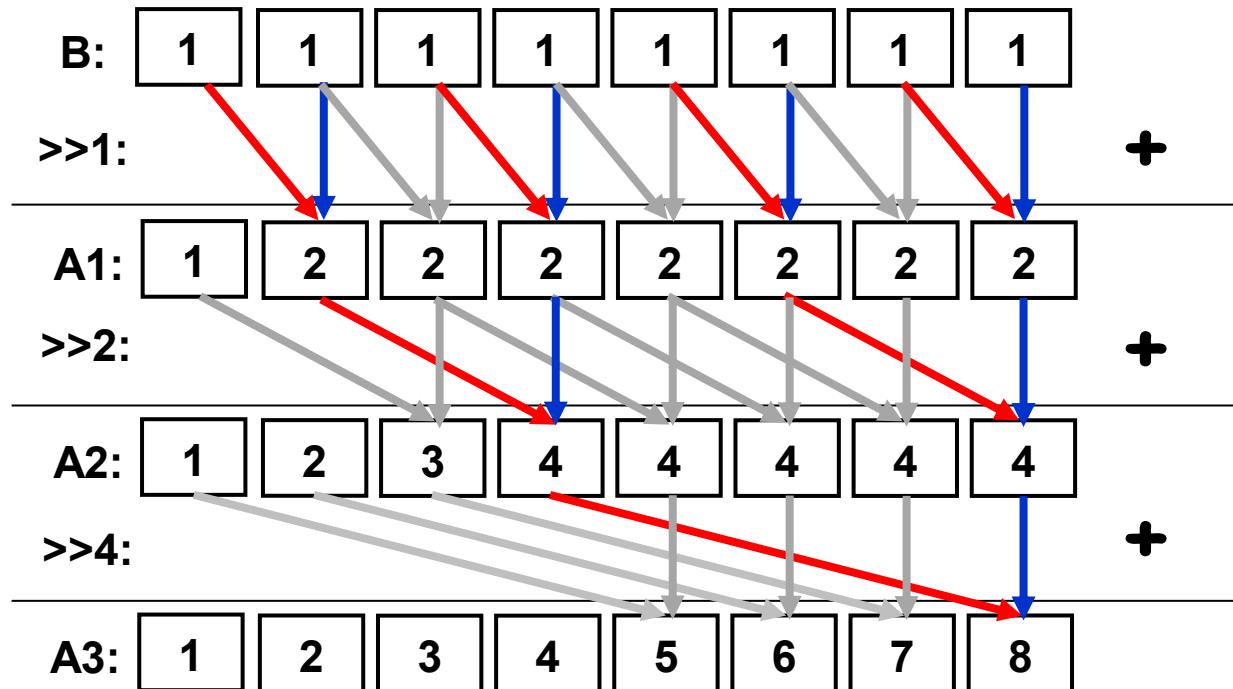
for i = 1 to 8

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

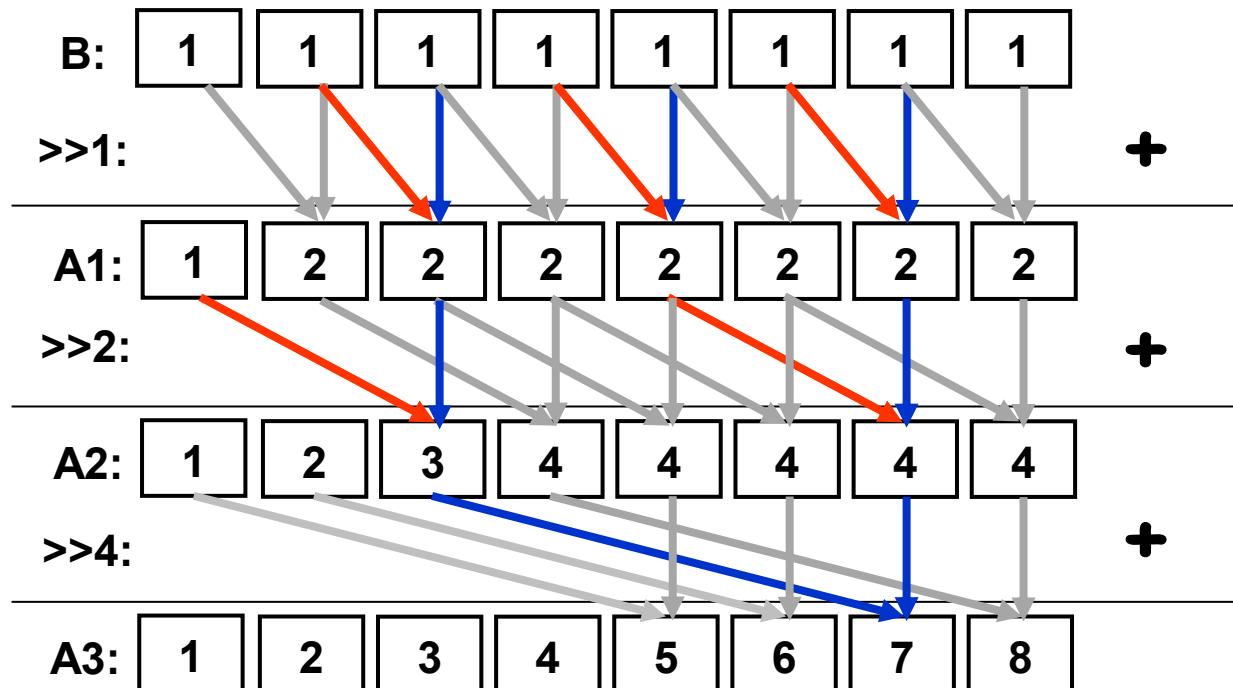
for i = 1 to 8

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

S1 : A[0] := 0

for i = 1 to 8

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

Feeding curiosity

↳ Appears to be inherently sequential

↳ But parallel implementation is possible

B:	1	1	1	1	1	1	1	1
>>1:	1	1	1	1	1	1	1	+
<hr/>								
B:	1	2	2	2	2	2	2	
>>2:	1	2	2	2	2	2	2	+
<hr/>								
B:	1	2	3	4	4	4	4	
>>4:	1	2	3	4	+			
<hr/>								
B:	1	2	3	4	5	6	7	8

“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>