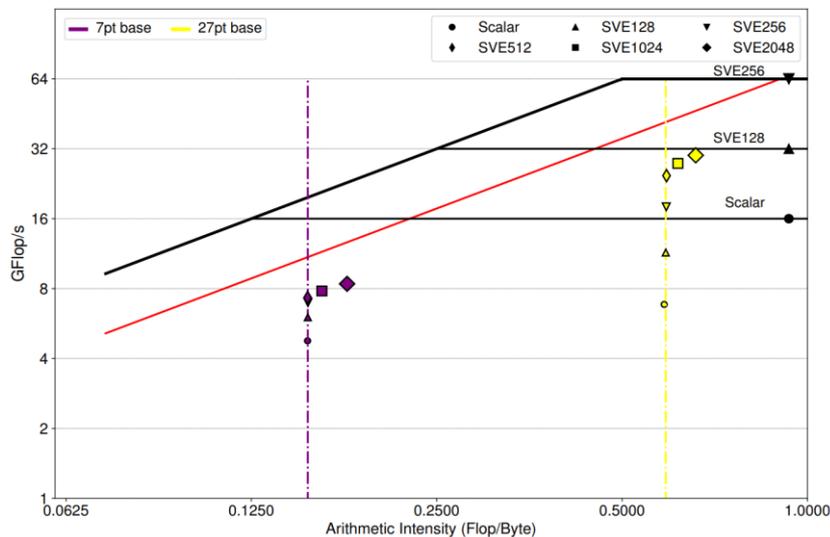


Advanced Computer Architecture

Chapter 8:

Vectors, vector instructions, vectorization and SIMD



November 2023
Paul H J Kelly

This section has contributions from Fabio Luporini (PhD & postdoc at Imperial, now CTO of DevitoCodes) and Luigi Nardi (ex Imperial and Stanford postdoc, now an academic at Lund University).

Course materials online at

<http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture.html>

- ▶ 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?

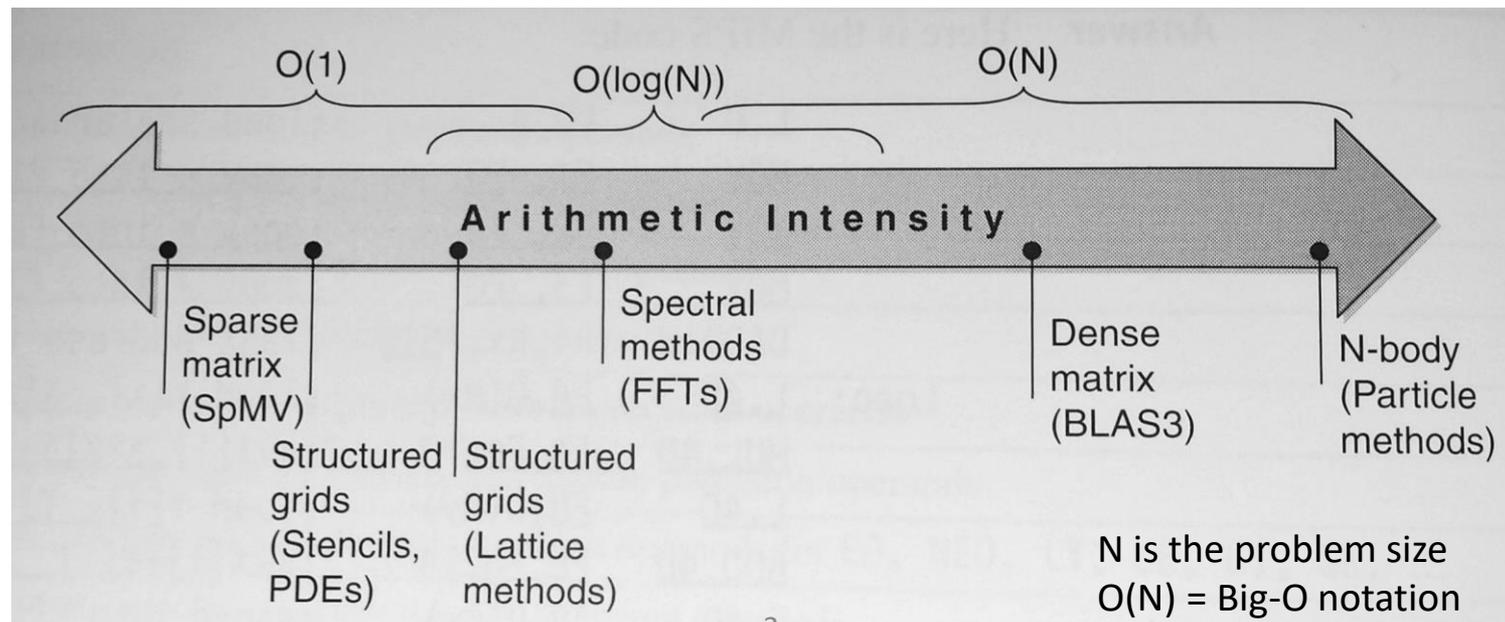
- ▶ And then, in the next chapter: GPUs, and Single-Instruction Multiple Threads (SIMT)

Arithmetic Intensity

	Processor	Type	Peak GFLOP/s	Peak GB/s	Ops/Byte	Ops/Word
Intel	E5-2690 v3* SP	CPU	416	68	~6	~24
	E5-2690 v3 DP	CPU	208	68	~3	~24
NVIDIA	K40** SP	GPU	4,290	288	~15	~60
	K40 DP	GPU	1,430	288	~5	~40

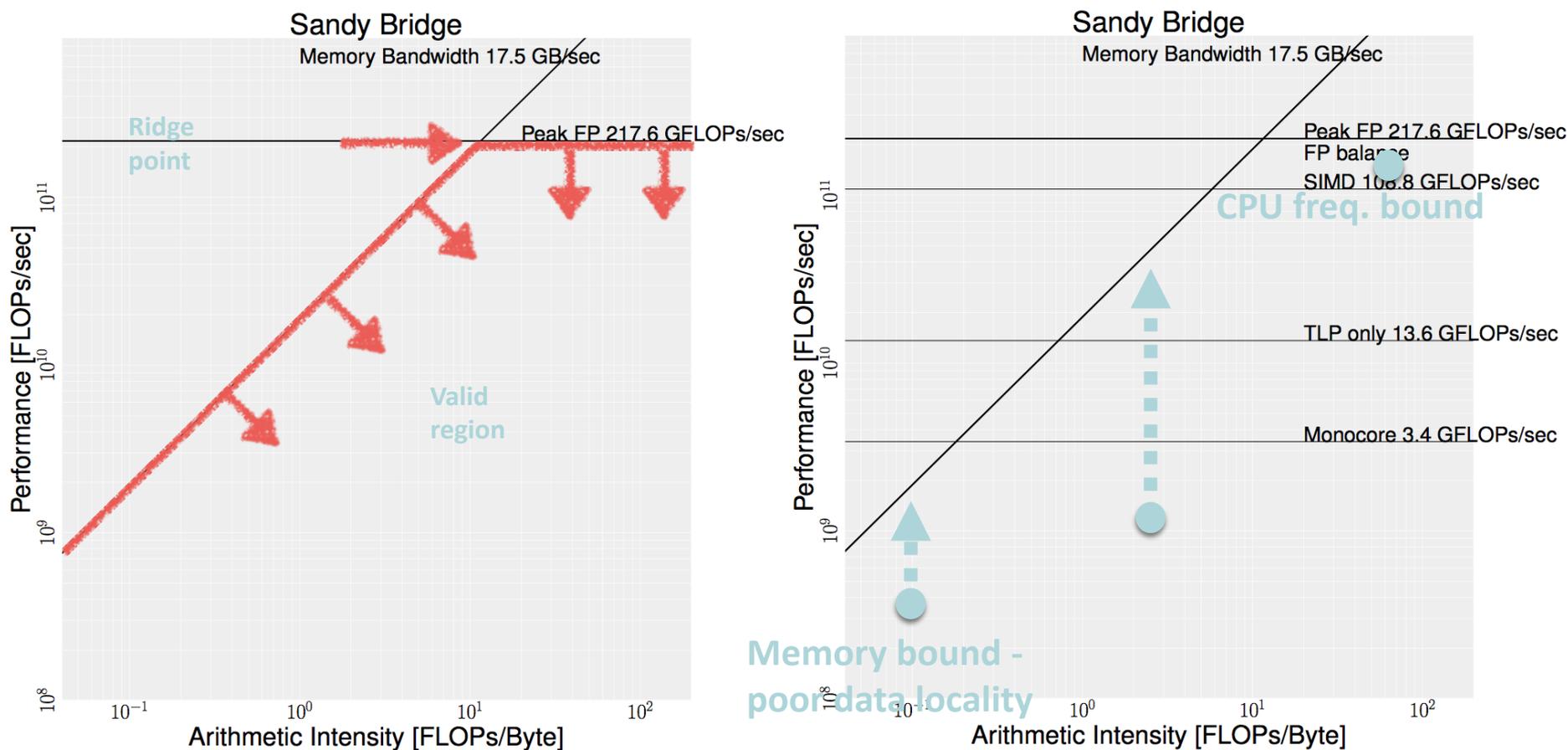
If the hardware has high Ops/Word, some code is likely to be bound by operand delivery (SP: single-precision, 4B/word; DP: double-precision, 8B/word)

Arithmetic intensity: Ops/Byte of DRAM traffic

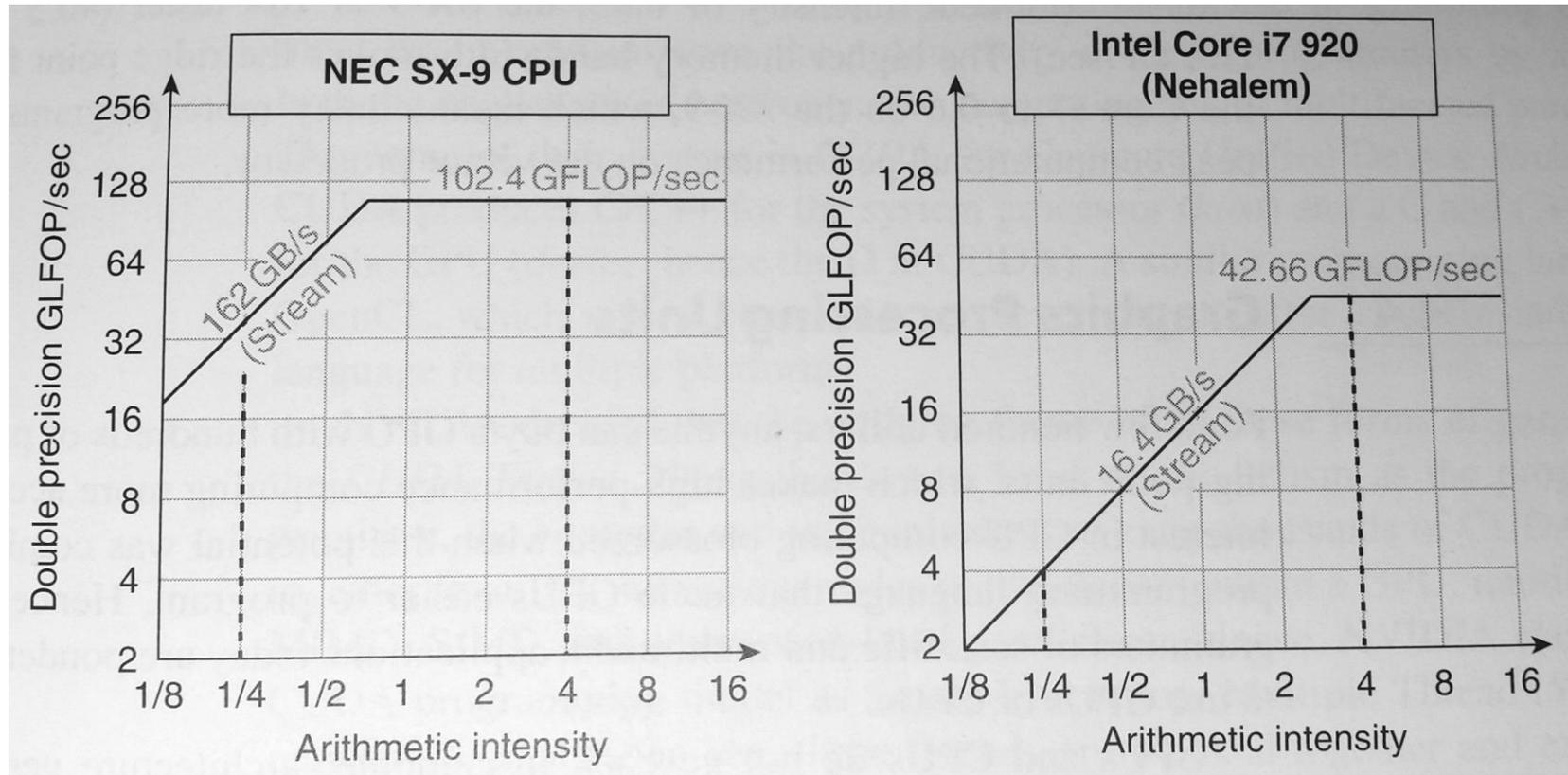


Roofline Model: Visual Performance Model

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



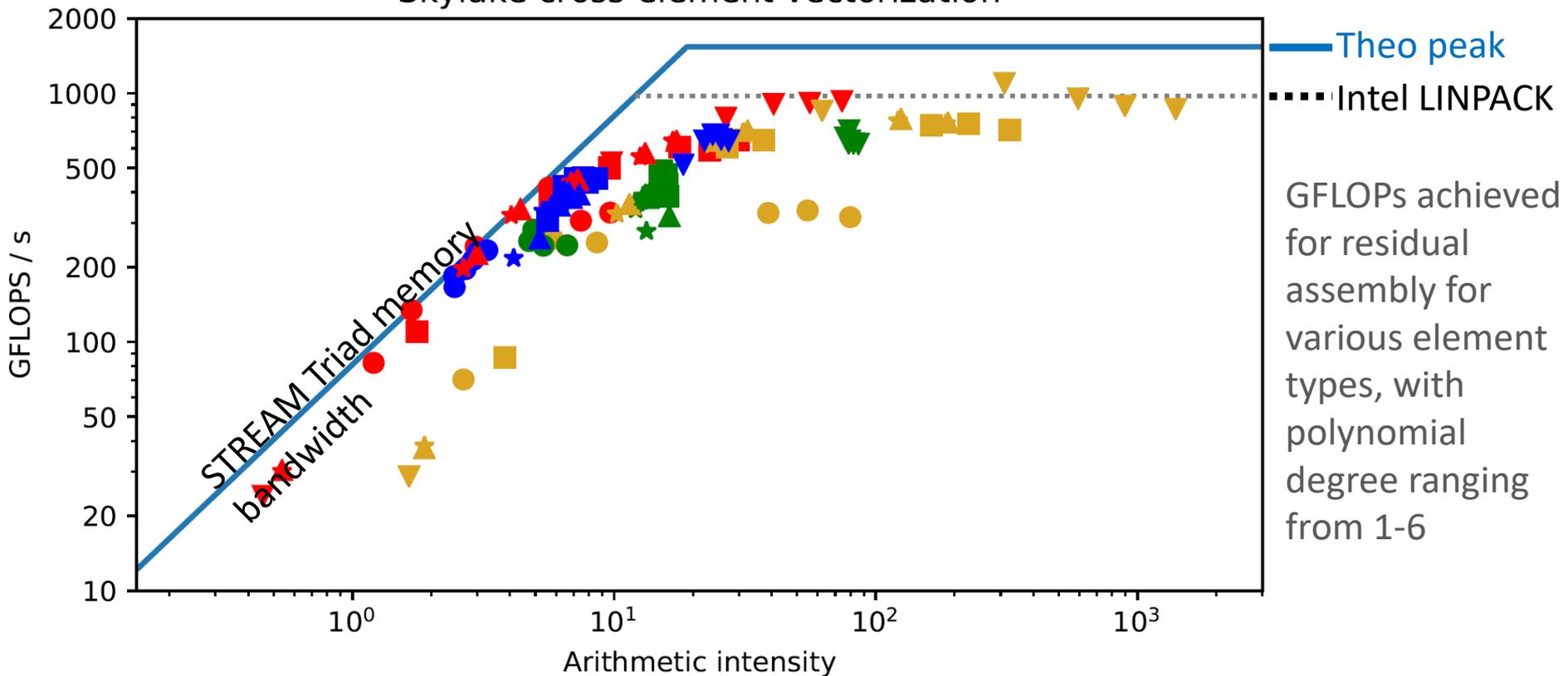
Roofline Model: Visual Performance Model



- The 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

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

Skylake cross-element vectorization



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

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 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?

Compiler Explorer

Editor Diff View More Share Other

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

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 -O3 -fopenmp

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

```

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
111010 .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     cmpl $2, %ecx
9     leal 0(%rdx,4), %eax
10    jbe .L9
11    xorl %ecx, %ecx
12    xorl %esi, %esi
13
14 .L5:
15    movaps a(%rcx), %xmm0
16    addl $1, %esi
17    addq $16, %rcx
18    addps b-16(%rcx), %xmm0
19    movaps %xmm0, c-16(%rcx)
20    cmpl %esi, %edx
21    ja .L5
22    cmpl %edi, %eax
23    je .L12
24
25 .L3:
26    movslq %eax, %rdx
27    movss b(,%rdx,4), %xmm0
28    addss a(,%rdx,4), %xmm0
29    movss %xmm0, c(,%rdx,4)
30    leal 1(%rax), %edx
31    cmpl %edx, %edi
32    jle .L1
33    movslq %edx, %rdx
34    addl $2, %eax
35    movss a(,%rdx,4), %xmm0
36    cmpl %eax, %edi
37    addss b(,%rdx,4), %xmm0
38    movss %xmm0, c(,%rdx,4)
39    jle .L1
40    cltq
41    movss a(,%rax,4), %xmm0
42    addss b(,%rax,4), %xmm0
43    movss %xmm0, c(,%rax,4)
44    ret
45
46 .L1:
47    rep ret
48
49 .L12:
50    rep ret
51
52 .L9:
53    xorl %eax, %eax
54    jmp .L3
55
56 b:
57     .zero 4096
58
59 a:
60     .zero 4096
61
62 c:
63     .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

```

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
1 23ad0f5_5_1:
2 testl %eax, %eax
3 pushq %r13
4 pushq %r12
5 pushq %rbp
6 pushq %rbx
7 js .L1
8 mov %r13, %rax
9 leal 1(%r0), %r0d
10 andl $15, %eax
11 shrq $2, %rax
12 rorq %rax
13 andl $3, %eax
14 cmpi %r0d, %eax
15 cmova %r0d, %eax
16 cmpi $4, %r0d
17 je .L2
18 movl %r0d, %eax
19
20 .L3:
21 movss (%r1), %xmm0
22 cmpi $1, %eax
23 movl $1, %r0d
24 addss (%r0), %xmm0
25 movss %xmm0, (%r1)
26 je .L5
27 movss 4(%r1), %xmm0
28 cmpi $2, %eax
29 movl $2, %r0d
30 addss 4(%r0), %xmm0
31 movss %xmm0, 4(%r1)
32 je .L5
33 movss 8(%r1), %xmm0
34 cmpi $3, %eax
35 movl $3, %r0d
36 addss 8(%r0), %xmm0
37 movss %xmm0, 8(%r1)
38 je .L5
39 movss 12(%r1), %xmm0
40 movl $4, %r0d
41 addss 12(%r0), %xmm0
42 movss %xmm0, 12(%r1)
43
44 .L5:
45 cmpi %eax, %r0d
46 je .L1
47
48 .L4:
49 subl %eax, %r0d
50 movl %eax, %rbx
51 movl %eax, %r1d
52 leal -4(%r0), %r10d
53 subl %eax, %rbx
54 shrq $2, %r0d
55 addl $1, %r10d
56 cmpi $2, %rbx
57 leal 0(%r10,4), %rbp
58 jmp .L7
59 leaq 0(%r1,4), %rax
60 xorl %ebx, %ebx
61 leaq (%r1,%rax), %r13
62 leaq (%r0,%rax), %r12
63 leaq (%r1,%rax), %r11
64 xorl %eax, %eax
65
66 .L9:
67 movups (%r1,%rax), %xmm0
68 addl $1, %ebx
69 addps 0(%r1,%rax), %xmm0
70 movups %xmm0, (%r1,%rax)
71 addq $15, %rax
72 cmpi %ebx, %r10d
73 je .L9
74
75 .L7:
76 movsld %r0d, %rax
77 movss (%r1,%rax,4), %xmm0
78 addss (%r0,%rax,4), %xmm0
79 movss %xmm0, (%r1,%rax,4)
80 leal 1(%r0), %eax
81 cmpi %eax, %ecx
82 jl .L1
83
84 .L10:
85 addl $2, %r0d
86 movss (%r1,%rax,4), %xmm0
87 cmpi %r0d, %ecx
88 addss (%r0,%rax,4), %xmm0
89 movss %xmm0, (%r1,%rax,4)
90 jl .L1
91
92 movsld %r0d, %r0
93 movss (%r1,%r0,4), %xmm0
94 addss (%r0,%r0,4), %xmm0
95 movss %xmm0, (%r1,%r0,4)
96
97 .L11:
98 popq %rbx
99 popq %rbp
100 popq %r12
101 popq %r13
102 ret
103
104 .L2:
105 testl %eax, %eax
106 jmp .L3
107
108 movl %r0d, %r0d
109 jmp .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

```

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 }

```

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

x86-64 gcc 5.4 (Editor #1, Compiler #1) C++ x -O3 -fopt-info

```

A- 11010 LXX text // ls+ Intel Demangle Libraries + Add new...
1 123456789_5_5
2 testl %eax, %eax
3 jle .L27
4 leaq 16(%rax), %rax
5 leaq 16(%rax), %rax
6 leal 16(%rax), %rax
7 cmov %rax, %rax
8 setno %rax
9 cmov %rax, %rax
10 setno %rax
11 ori %eax, %rax
12 leaq 16(%rax), %rax
13 cmov %rax, %rax
14 setno %rax
15 cmov %rax, %rax
16 setno %rax
17 ori %rax, %rax
18 testb %al, %rbp
19 jle .L3
20 cmpl $4, %rax
21 jbe .L3
22 movq %rax, %rax
23 pushq %rax
24 pushq %rax
25 andl $15, %eax
26 pushq %rax
27 pushq %rax
28 shrq $2, %rax
29 raddq %rax
30 andl $1, %eax
31 cmov %rax, %rax
32 cmovb %rbp, %rax
33 xorl %rax, %rax
34 testl %eax, %eax
35 je .L4
36 movss (%rax), %xmm0
37 cmovl $1, %rax
38 movl $1, %rax
39 addss (%rax), %xmm0
40 movss %xmm0, (%rax)
41 je .L4
42 movss 4(%rax), %xmm0
43 cmovl $1, %rax
44 movl $1, %rax
45 addss 4(%rax), %xmm0
46 movss %xmm0, 4(%rax)
47 je .L4
48 movss 8(%rax), %xmm0
49 cmovl $1, %rax
50 movl $1, %rax
51 movss %xmm0, 8(%rax)
52 .L4:
53 subl %eax, %rax
54 shld %2, %rax
55 xorl %rax, %rax
56 leal -4(%rax), %rax
57 leaq (%rax,%rax), %rax
58 leaq (%rax,%rax), %rax
59 xorl %eax, %eax
60 addq %rax, %rax
61 shrl $2, %rax
62 andl $1, %rax
63 leal 0(%rax), %rax
64 .L7:
65 movq (%rax,%rax), %xmm0
66 addl $1, %rax
67 addq 0(%rax,%rax), %xmm0
68 movq %xmm0, (%rax,%rax)
69 addl $4, %rax
70 cmpl %rax, %rax
71 jle .L7
72 addl %rax, %rax
73 cmpl %rax, %rax
74 jle .L7
75 movl %rax, %rax
76 movq (%rax,%rax), %xmm0
77 addq (%rax,%rax), %xmm0
78 movq %xmm0, (%rax,%rax)
79 leal 16(%rax), %rax
80 cmov %rax, %rax
81 jle .L11
82 cmov %rax, %rax
83 movq (%rax,%rax), %xmm0
84 cmov %rax, %rax
85 cmov %rax, %rax
86 addq (%rax,%rax), %xmm0
87 movq %xmm0, (%rax,%rax)
88 jle .L11
89 movq %rax, %rax
90 movq (%rax,%rax), %xmm0
91 addq (%rax,%rax), %xmm0
92 movq %xmm0, (%rax,%rax)
93 .L11:
94 pushq %rax
95 pushq %rax
96 pushq %rax
97 pushq %rax
98 .L17:
99 rep ret
100 .L13:
101 xorl %eax, %eax
102 .L12:
103 movq (%rax,%rax), %xmm0
104 addq (%rax,%rax), %xmm0
105 movq %xmm0, (%rax,%rax)
106 addq $1, %rax
107 cmov %rax, %rax
108 jge .L12
109 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

Output (0/7) g++ (GCC-Explorer-Build) 5.4.0 - 464ms (7111B)

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**

loopwise:

```
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)

functionwise:

```
#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 #2: 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 #3: SIMT¹⁷

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

```

1 // icc: -xCORE-AVX512 -qopt-zmm-usage=high -qo
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 -xCORE-AVX512 -qopt-zmm-usage=h

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 zmm0, 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)

```

C source #1 x
A Save/Load Add new... Vim C
1 // icc: -xCORE-AVX512 -qopt-zmm-usage=high -qo
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=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 [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??)



Add... More

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

Share Other Policies

C source #1 x

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

Save/Load Add new... Vim C

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

```

1 // icc: -xCORE-AVX512 -qopt-zmm-usage=high -qo
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 }

```

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 x

Wrap lines

Compiler returned: 0

```

1 // icc: -xCORE-AVX512 -qopt-zmm-usage=high -qo
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 x
x86-64 icc 19.0.1 -xCORE-AVX512 -qopt-zmm-usage=h
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

Vector execution alternatives

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

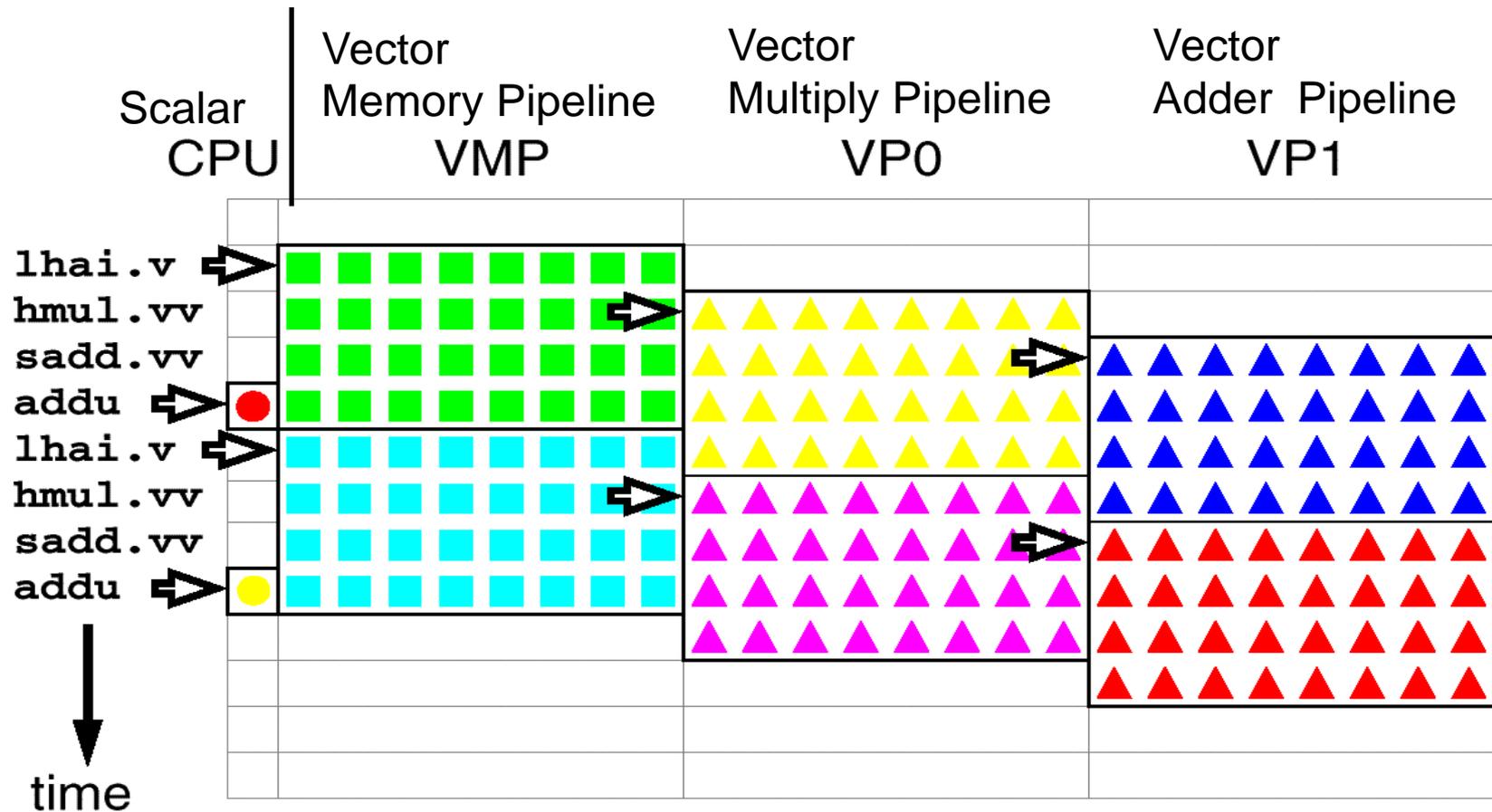
↳ **vector pipelining:**

- 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”



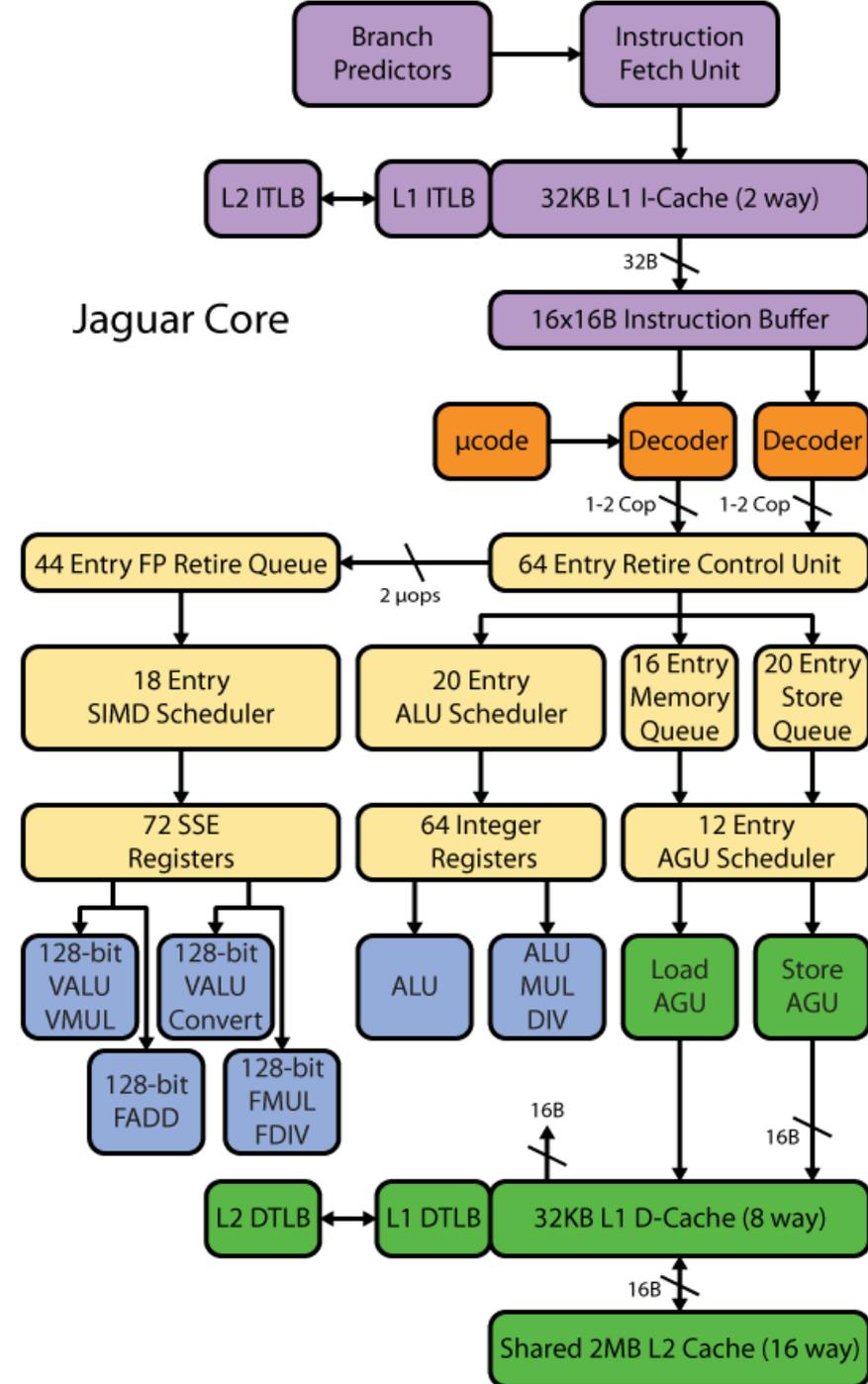
● ■ ▲ Operations
 ➡ Instruction issue

- 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, RISC-V 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, SIMT**

- ▶ Vectorising nested conditionals
- ▶ Vectorising non-innermost loops
- ▶ Vectorising loops containing while loops

▶ **SIMT 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

```

```

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

```

```

1  f(char*, char*):                       # @f(char*, char*)
2      vmovups ymm0, ymmword ptr [rsi]
3      vmovups ymmword ptr [rdi], ymm0
4      vzeroupper
5      ret

```