332
Advanced Computer Architecture
Chapter 7.1:

Vectors, vector instructions, vectorization and SIMD

March 2019

Paul H J Kelly

This section has contributions from Fabio Luporini (postdoc at Imperial) and Luigi Nardi (now a postdoc at Stanford).

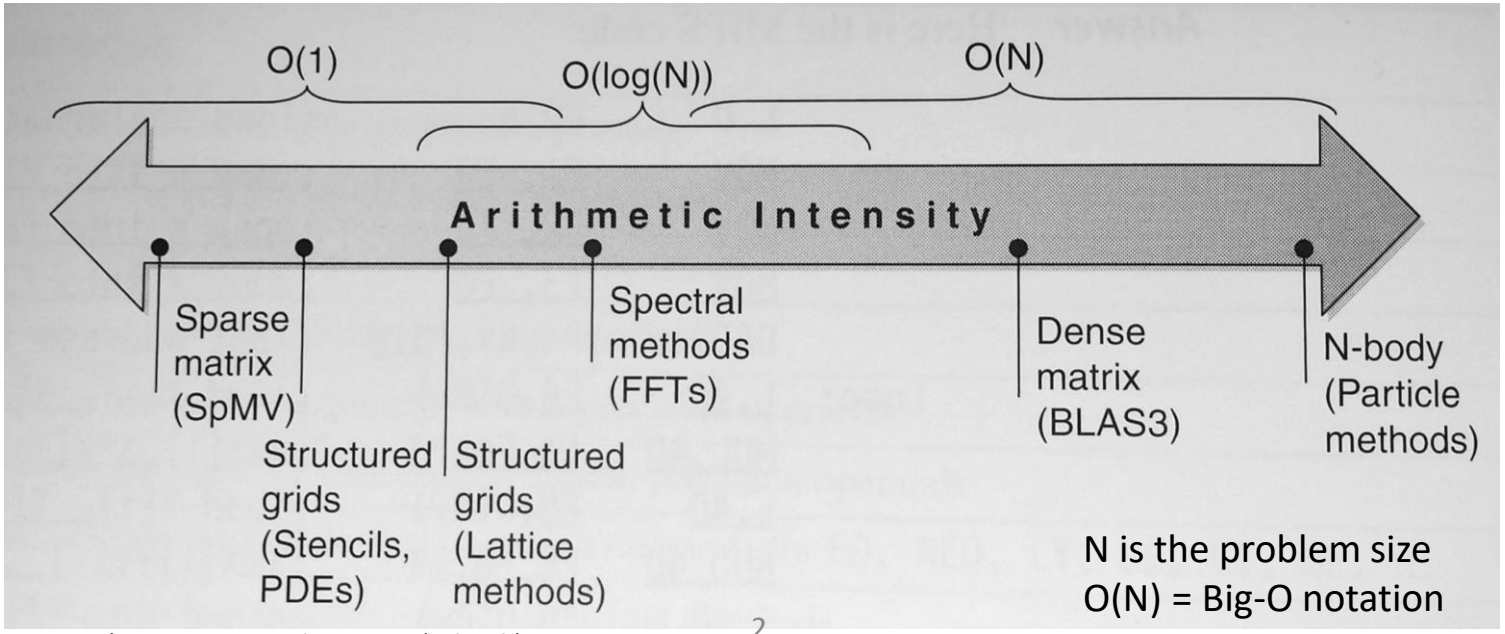
Course materials online at
http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture.html

Arithmetic Intensity

E5-2690 v3 SP	CPU	416	68	~6	~24
E5-2690 v3 DP	CPU	208	68	~3	~24
K40 SP	GPU	4,290	288	~15	~60
K40 DP	GPU	1,430	288	~5	~40

Without enough Ops/Word codes are likely to be bound by operand delivery (SP: single-precision, 4B/word; DP: double-precision, 8B/word)

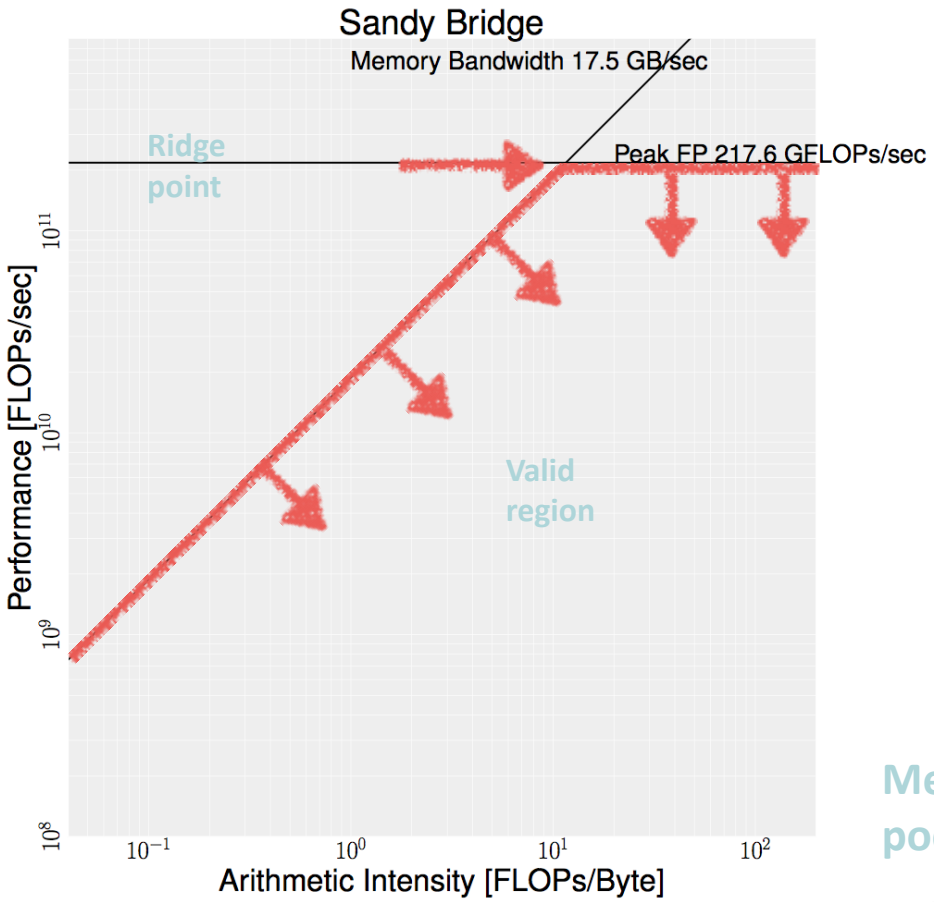
Arithmetic intensity: Ops/Byte of DRAM traffic

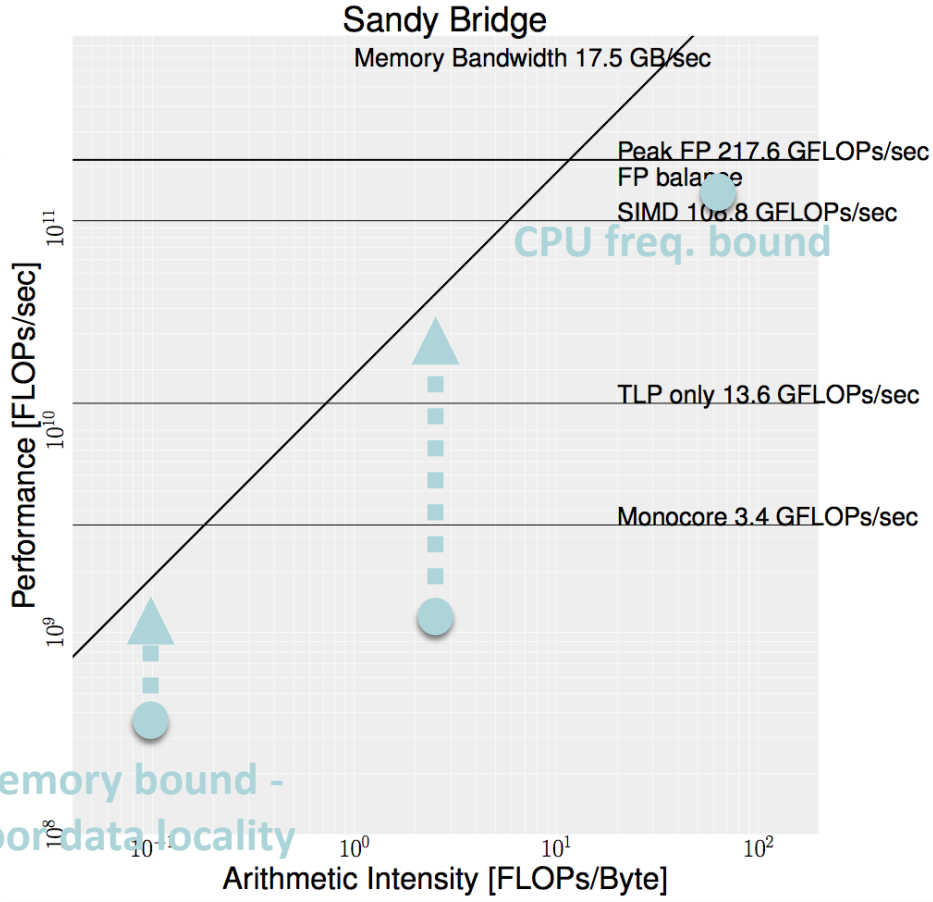


Hennessy and Patterson's Computer Architecture (5th ed.)

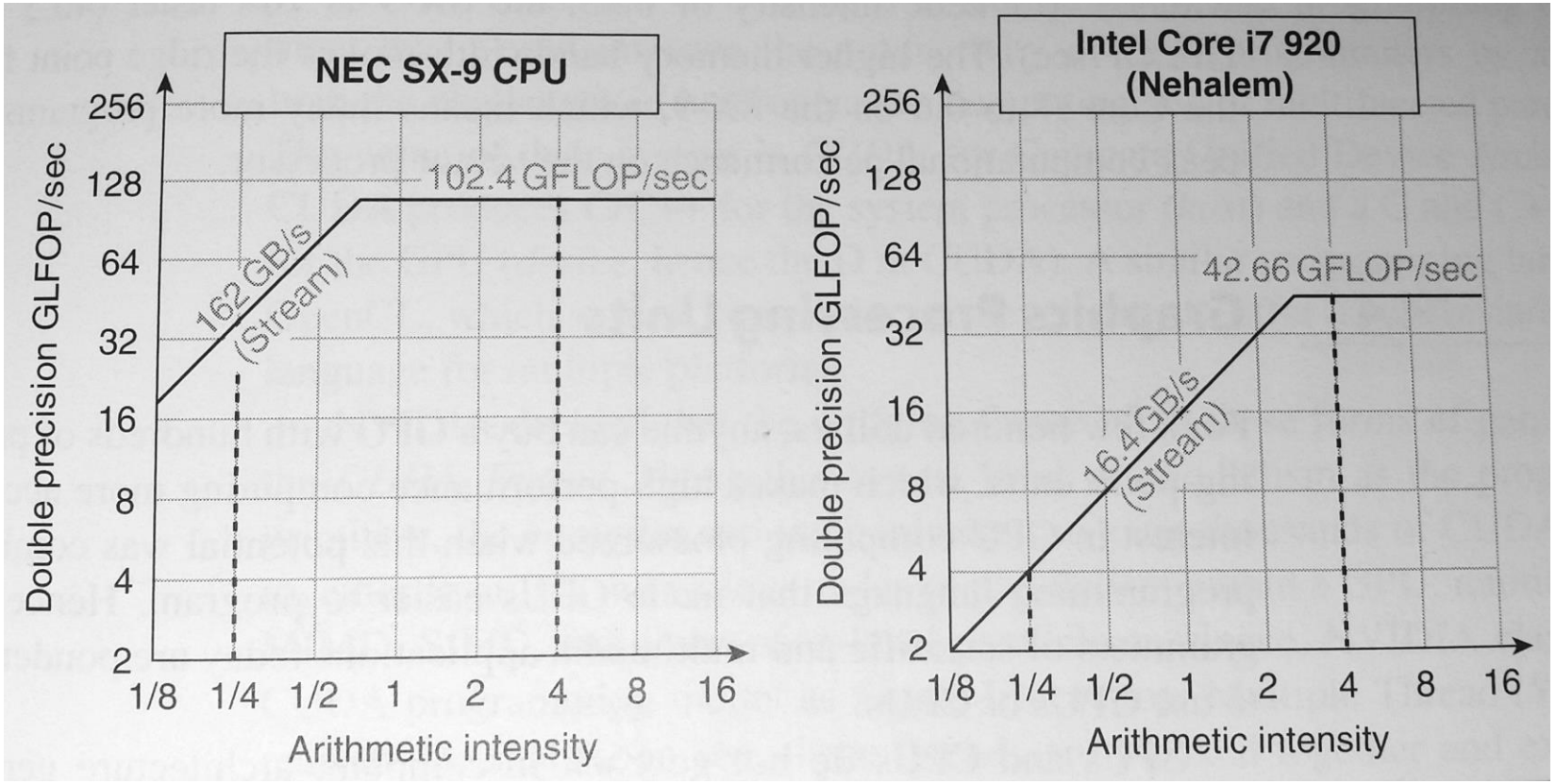
Roofline Model: Visual Performance Model

- 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* limited by memory bandwidth, or by arithmetic capability**

Hennessy and Patterson's Computer Architecture (5th ed.)

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 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

AVX512: vector addition

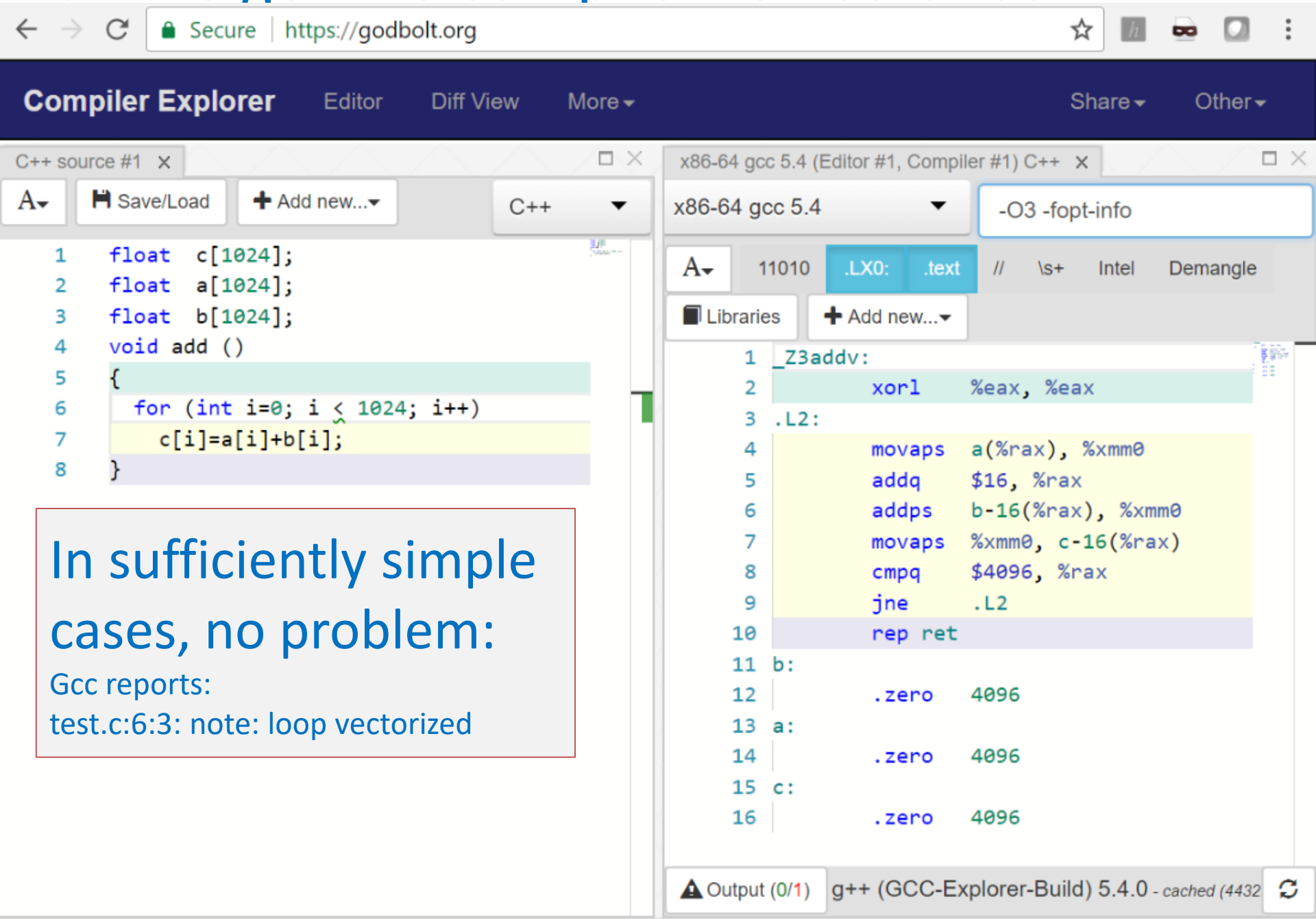
- Assembler:
  - VADDPS zmm1 {k1}{z}, zmm2, zmm3
- In C the compiler provides "vector intrinsics" that enable you to emit specific vector instructions, eg:
  - res = _mm512_maskz_add_ps(k, a, b);
- Only lanes with their corresponding bit in k1 are activated
- Two predication modes: *masking* and *zero-masking*
  - 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←0 TO KL-1
   i←j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i]←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] ← 0
         FI
   FI;
ENDFOR;
```

http://www.felixcloutier.com/x86/ADDPS.html

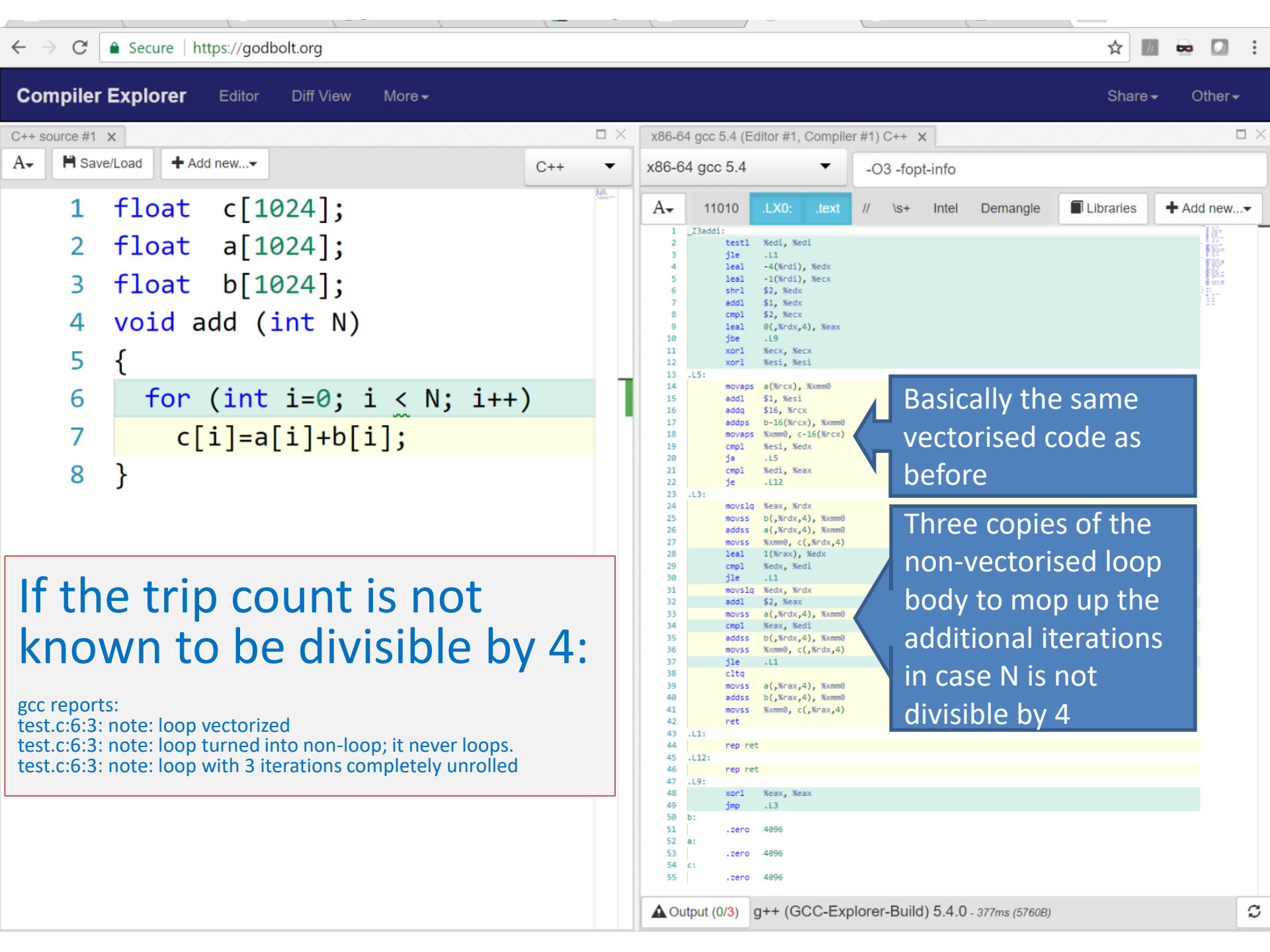
Can we get the compiler to vectorise?

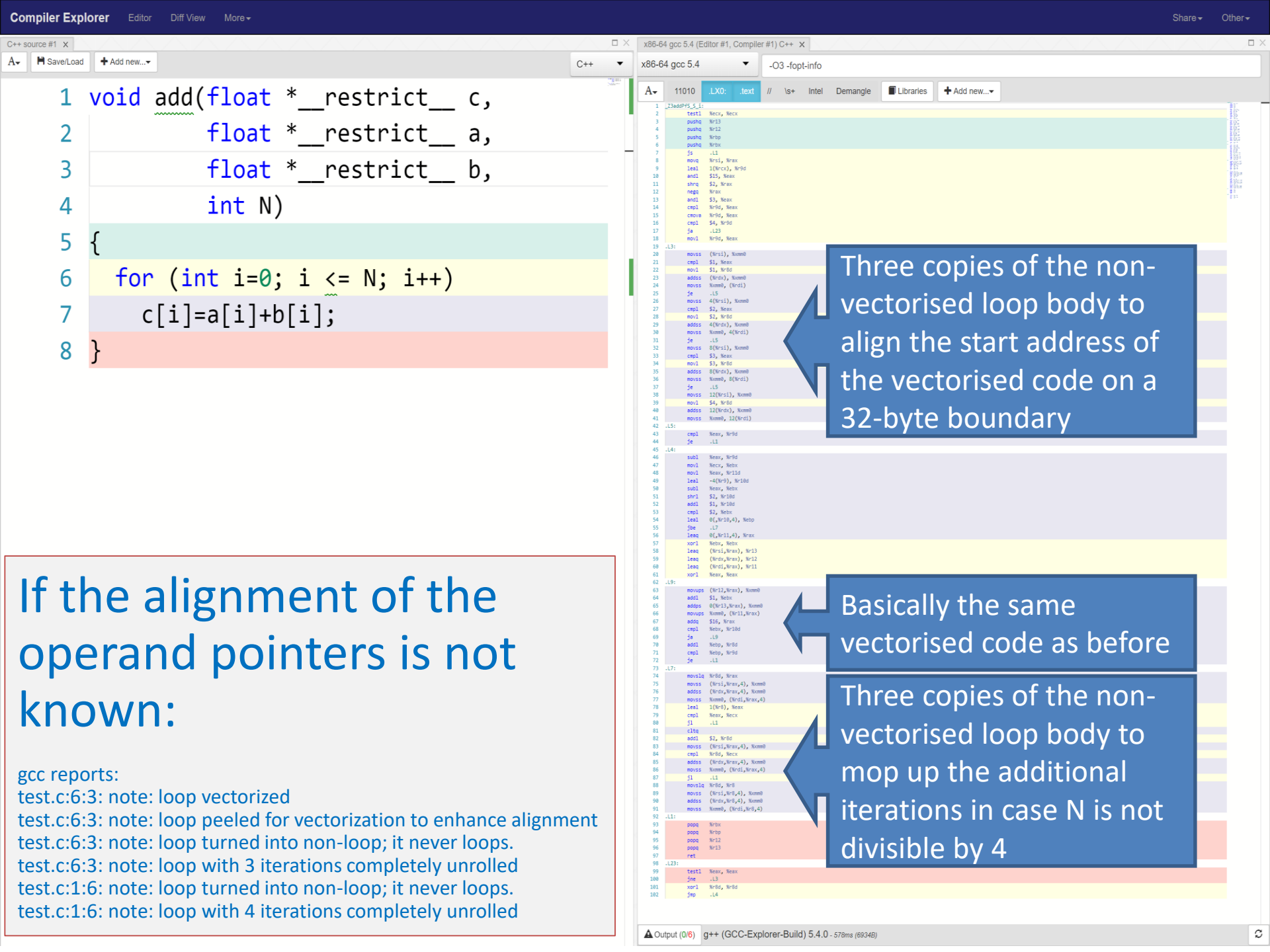


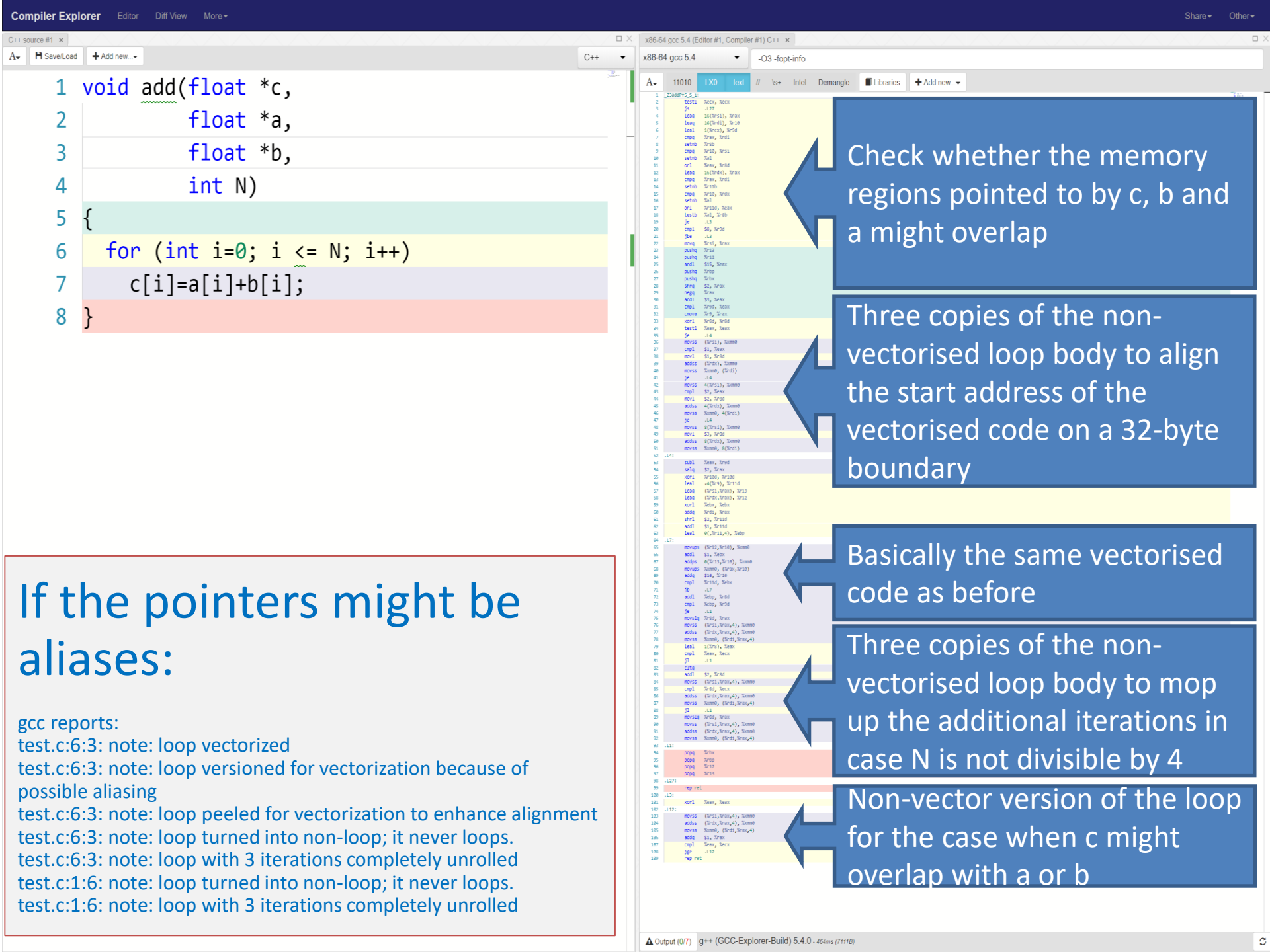
In sufficiently simple cases, no problem:

Gcc reports:

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


Secure https://godbolt.org
Compiler Explorer
Editor
Diff View
More
Share
Other
C++ source #1
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++
x86-64 gcc 5.4
-O3 -fopt-info
11010
.LX0:
.text
Intel
Demangle
Libraries
Add new...
_Z3addi:
testl %edi, %edi
jle .L1
leal -4(%rdi), %edx
leal -1(%rdi), %ecx
shrl $2, %edx
addl $1, %edx
cmpl $2, %ecx
leal 0(,%rdx,4), %eax
jbe .L9
xorl %ecx, %ecx
xorl %esi, %esi
.L5:
movaps a(%rcx), %xmm0
addl $1, %esi
addq $16, %rcx
addps b-16(%rcx), %xmm0
movaps %xmm0, c-16(%rcx)
cmpl %esi, %edx
ja .L5
cmpl %edi, %eax
je .L12
.L3:
movslq %eax, %rdx
movss b(,%rdx,4), %xmm0
addss a(,%rdx,4), %xmm0
movss %xmm0, c(,%rdx,4)
leal 1(%rax), %edx
cmpl %edx, %edi
jle .L1
movslq %edx, %rdx
addl $2, %eax
movss a(,%rdx,4), %xmm0
cmpl %eax, %edi
addss b(,%rdx,4), %xmm0
movss %xmm0, c(,%rdx,4)
jle .L1
cltq
movss a(,%rax,4), %xmm0
addss b(,%rax,4), %xmm0
movss %xmm0, c(,%rax,4)
ret
.L1:
rep ret
.L12:
rep ret
.L9:
xorl %eax, %eax
jmp .L3
b:
.zero 4096
a:
.zero 4096
c:
.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)


Compiler Explorer
Editor
Diff View
More
Share
Other
C++ source #1
Save/Load
Add new...
C++
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++
x86-64 gcc 5.4
-O3 -fopt-info
Libraries
Add new...
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
Output (0/6)
g++ (GCC-Explorer-Build) 5.4.0


Compiler Explorer
Editor
Diff View
More
Share
Other
C++ source #1
Save/Load
Add new...
C++
x86-64 gcc 5.4 (Editor #1, Compiler #1) C++
x86-64 gcc 5.4
-O3 -fopt-info
Intel
Demangle
Libraries
Add new...
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
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

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…

Source: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

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: SIMT

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

More later – when we look at GPUs

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

History:
Intel x86 ISA extended with SIMD

		width	Int.	SP	DP
1997	MMX	64	✔		
1999	SSE	128	✔	✔(x4)	
2001	SSE2	128	✔	✔	✔(x2)
2004	SSE3	128	✔	✔	✔
2006	SSSE 3	128	✔	✔	✔
2006	SSE 4.1	128	✔	✔	✔
2008	SSE 4.2	128	✔	✔	✔
2011	AVX	256	✔	✔(x8)	✔(x4)
2013	AVX2	256	✔	✔	✔
future	AVX-512	512	✔	✔(x16)	✔(x8)

ATPESC 2014, James Reinders: http://extremecomputingtraining.anl.gov/files/2014/08/20140804-1030-1115-ATPESC-Argonne-Reinders.2.pdf

History:

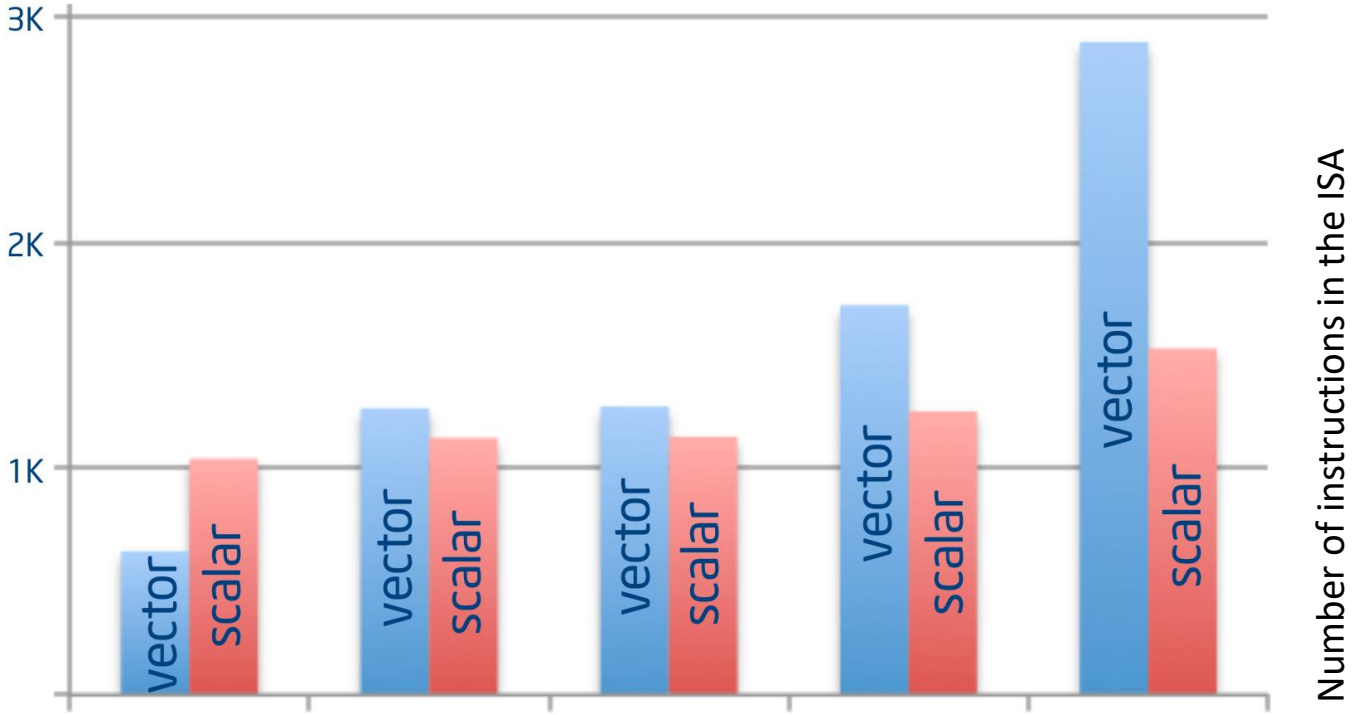
Intel x86 ISA ex

		width
1997	MMX	64
1999	SSE	128
2001	SSE2	128
2004	SSE3	128
2006	SSSE 3	128
2006	SSE 4.1	128
2008	SSE 4.2	128
2011	AVX	256
2013	AVX2	256
future	AVX-512	512

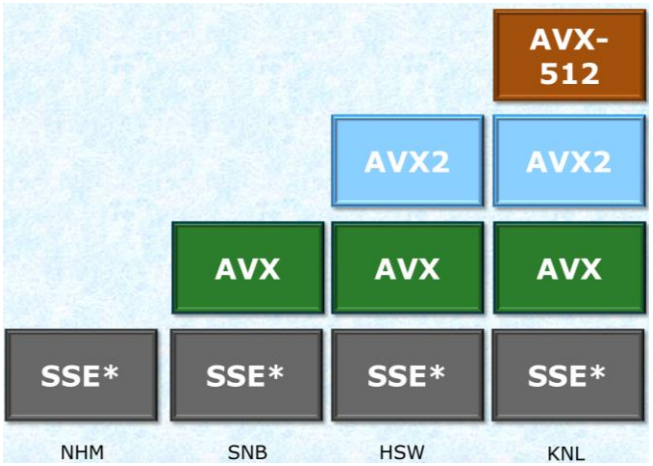
- **Wider registers**
  (from 32 to 512 bits)
- **More registers**
- **Richer instruction set**
  (predication, FMAs, gather, scatter, …)
- **Easier exploitation**
  (better compiler support, high-level functions, libraries…)

ATPESC 2014, James Reinders: http://extremecomputingtraining.anl.gov/files/2014/08/2014 …[iders.2.pdf](http://extremecomputingtraining.anl.gov/files/2014/08/2014)

Growth in vector instructions on Intel



Backwards compatibility accumulation



ATPESC 2014, James Reinders:
http://extremecomputingtraining.anl.gov/files/2014/08/20140804-1030-1115-ATPESC-Argonne-Reinders.2.pdf

Elena Demikhovsky (Intel): http://llvm.org/devmtg/2013-11/slides/Demikhovsky-Poster.pdf

Issues inherent in the vector model

Example 1

```
double A[N], B[N], C[N]
for i = 0 to N, i++
        C[i] = sqrt(A[i] + B[i])
```

SIMD version

```
loop: VLOAD a_v, A[i:v]
      VLOAD b_v, B[i:v]
      VADD c_v, b_v, a_v
      VSQRT c_v, c_v
      VSTORE C[i:v], c_v
      INCR i
      IF i<N/v: loop
```

Notation:

- **:v** indicates that the assembly operation is over v elements
- subscript **v** indicates that the register is actually a vector register, hosting **v** elements

E.g. **v**=4

Simple issues: bad array size

```
loop: VLOAD a_v, A[i:v]
      VLOAD b_v, B[i:v]
      VADD c_v, b_v, a_v
      VSQRT c_v, c_v
      VSTORE C[i:v], c_v
      INCR i
      IF i<N/v: loop
```

<u>Issue 1</u>: N might not be a multiple of the vector length v
or
N is known only at runtime

Simple issues: bad array size

```
loop: VLOAD av, A[i:v]
      VLOAD bv, B[i:v]
      VADD cv, bv, av
      VSQRT cv, cv
      VSTORE C[i:v], cv
      INCR i
      IF i<N/v: loop
      IF N%v==0: exit
peel: LOAD a, A[v*i + 0]
      …
      …
exit: …
```

<u>Issue 1</u>: N might not be a multiple of the vector length v

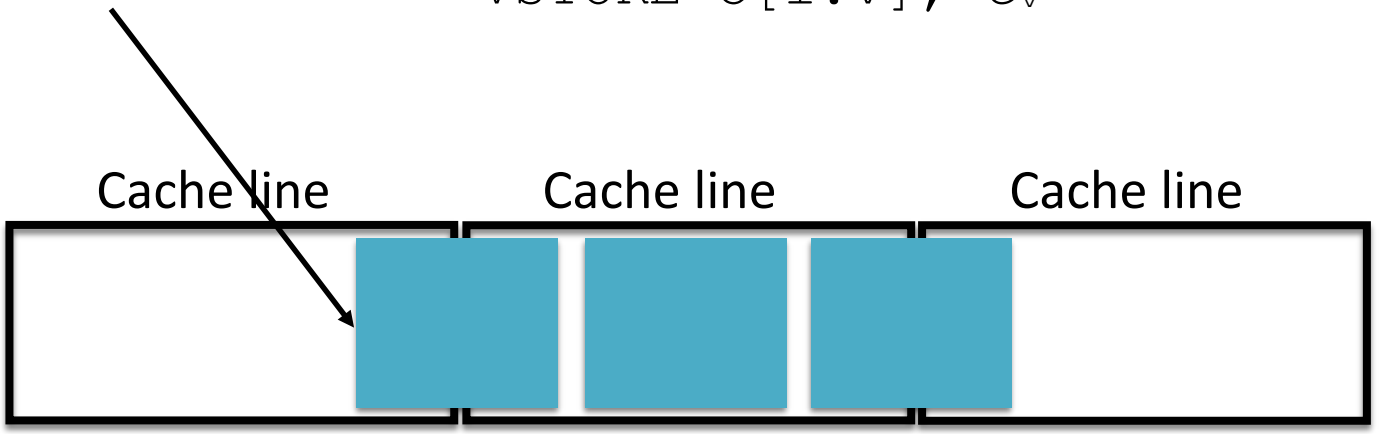
or

- Or not known at compile-time

Medium issues: data alignment

```
loop: VLOAD av, A[i:v]
      VLOAD bv, B[i:v]
      …
      VSTORE C[i:v], cv
```

Base address of array A



E.g.: AVX on Sandy Bridge: Cache line: 64B, vector length: 32B, double: 8B

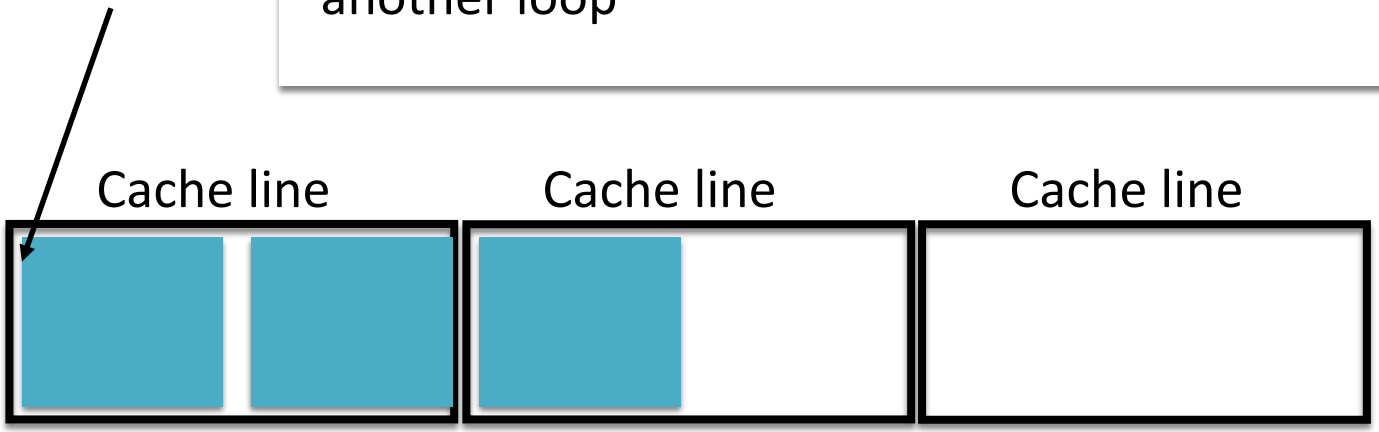
Issue 2: Memory accesses should be aligned to page and cache boundaries

Medium issues: data alignment

Solution: change the allocation point of A

- Use of special mallocs or special array qualifiers
- **Global transformation**: might affect alignment in another loop

Base address of array A



E.g.: AVX on Sandy Bridge: Cache line: 64B, vector length: 32B, double: 8B

Issue 2: Memory accesses should be aligned to page and cache boundaries
(tricky with stencils, for i {B[i]=A[i-1]+A[i]+A[i+1]})

Advanced issues: bad access patterns

Example 2

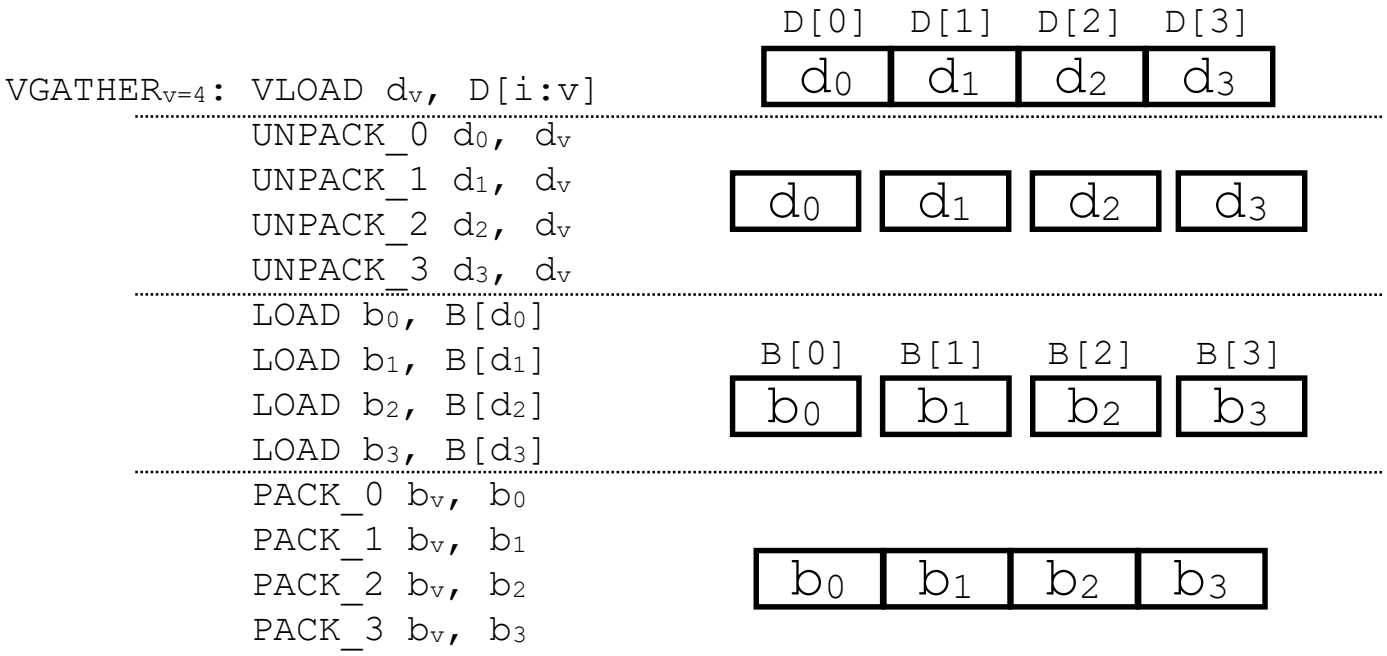
```
double A[N], B[N], C[N], D[N]
for i = 0 to N, i++
    C[i] = A[2*i] + B[D[i]]
```

SIMD version

```
loop: VLOAD a_v, A[i], stride=2
      VGATHER b_v, B, D[i:v]
      VADD c_v, b_v, a_v
      VSTORE C[i:v], c_v
incr: INCR i
      IF i<N/v: loop
```

Advanced issues: bad access patterns

```
B[D[i]]  ===>  VGATHER bv, B, D[i:v]
```



<u>Issue 3</u>: regardless of the ISA the (micro-)interpretation of these instructions is expensive

Each B[d_i] might fall on a different cache line, perhaps even a different page – vgather execution time depends on whether "coalescing" occurs

Advanced issues: branch divergence

```
for i = 0 to 63, i++
    if A[i] > 0
        B[i] = A[i]*4
```

```
loop:
  VLOAD      av, A[i:v]
  VCMP_P     Rmask, av, R0
  VMUL_P     bv{Rmask}, av, R4
  VSTORE_P   B[i:v]{Rmask}, bv
  VRESET_P   Rmask
  INCR       Ri
  CMP        Ri < 64/v: loop
```

Solution: **Predication through masking**

Add a new boolean vector register (the vector mask register)

- Operates on elements whose corresponding bit in the mask is 1
- Requires ISA extension to set the mask register

source: http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture11-vector.pdf

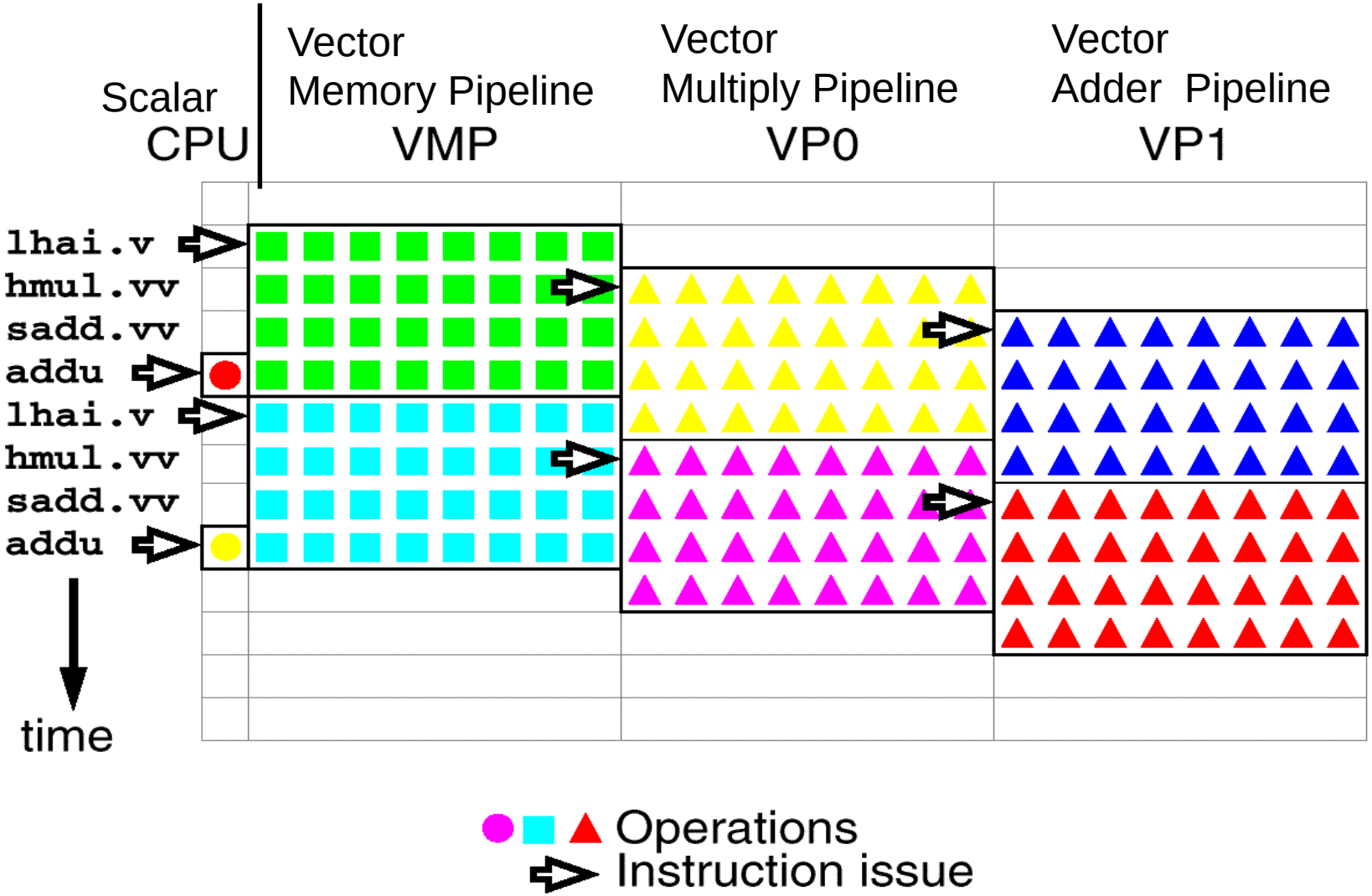
Vector execution alternatives

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**
  - **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**

source: http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture11-vector.pdf

Vector pipelining – "chaining"



- 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**

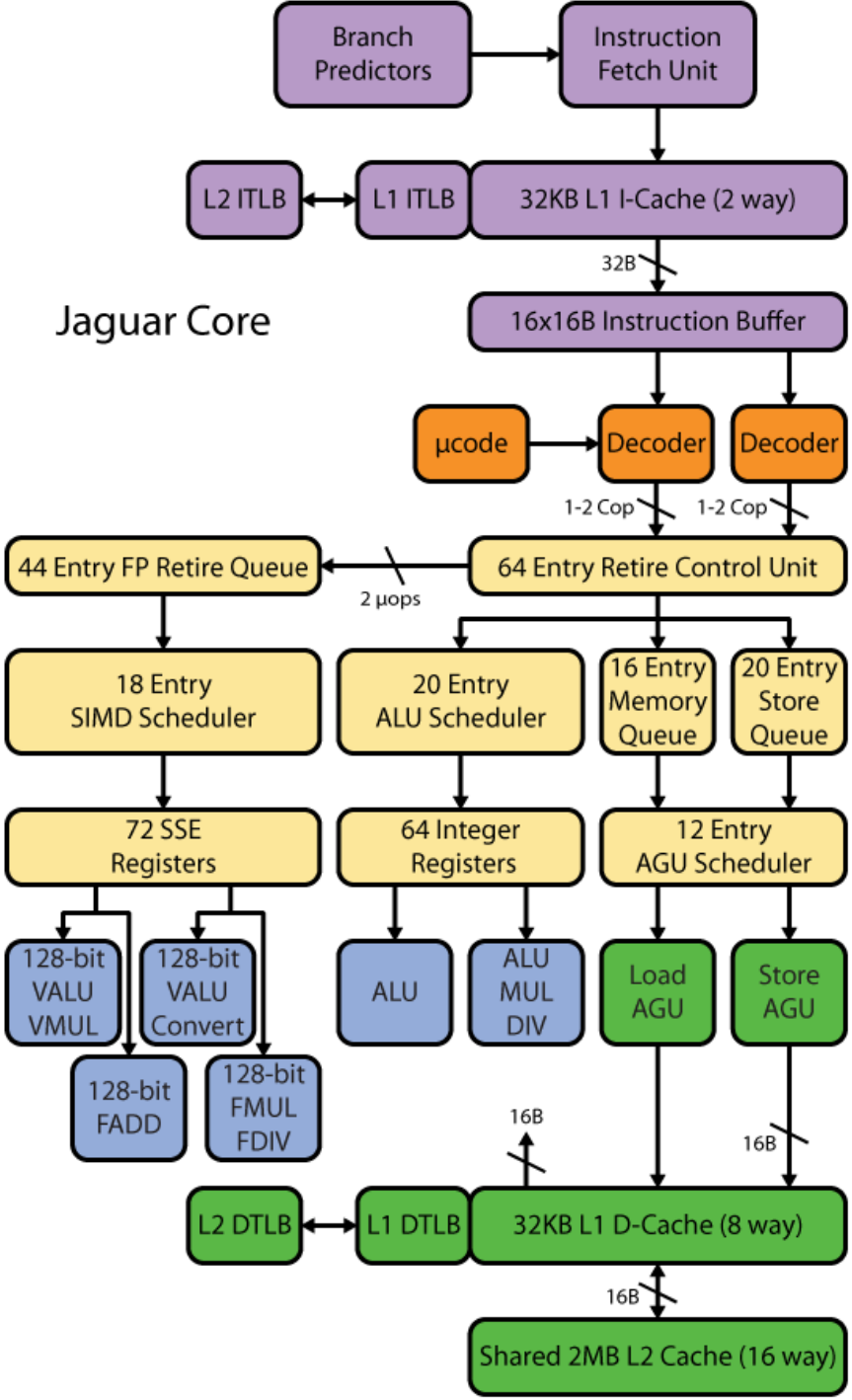


Figure from http://www.realworldtech.com/jaguar/4/ which includes more detail

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:**
  - **a vector ISA that achieves binary compatibility across machines with different vector width uop decomposition**
- **Matrix registers and matrix instructions**
  - **Eg Nvidia's "tensor cores"**
- **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!**