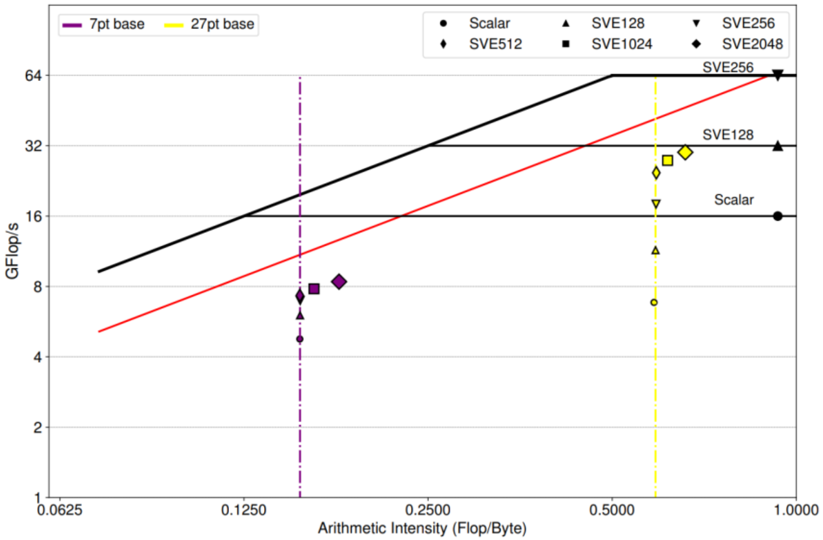
Advanced Computer Architecture

Chapter 8:

Vectors, vector instructions, vectorization and SIMD



November 2025

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 and founder at https://www.dbtune.com/).

Armejach, A., Caminal, H., Cebrian, J.M. *et al.* Using Arm's scalable vector extension on stencil codes. *J Supercomput* **76,** 2039–2062 (2020). https://doi.org/10.1007/s11227-019-02842-5

The plan

- **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?**

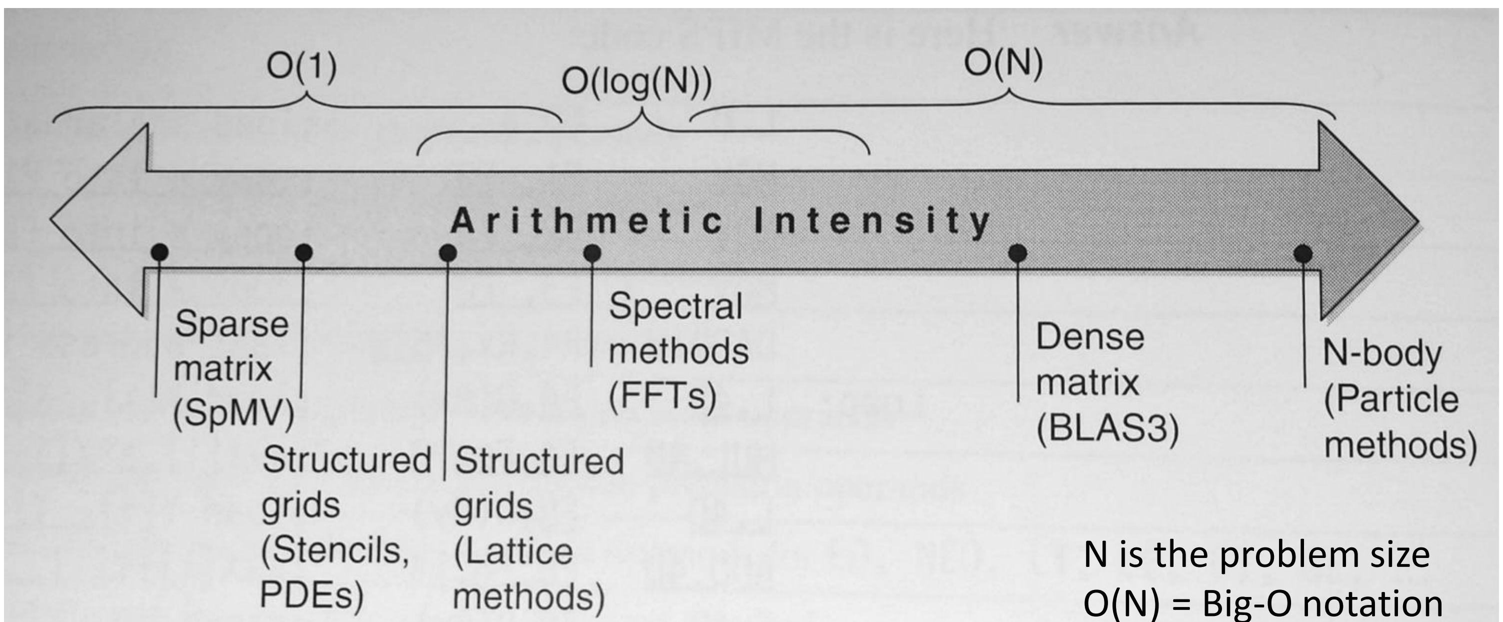
- **And then, in the next chapter: GPUs, and Single-Instruction Multiple Threads (SIMT)**

Arithmetic Intensity

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

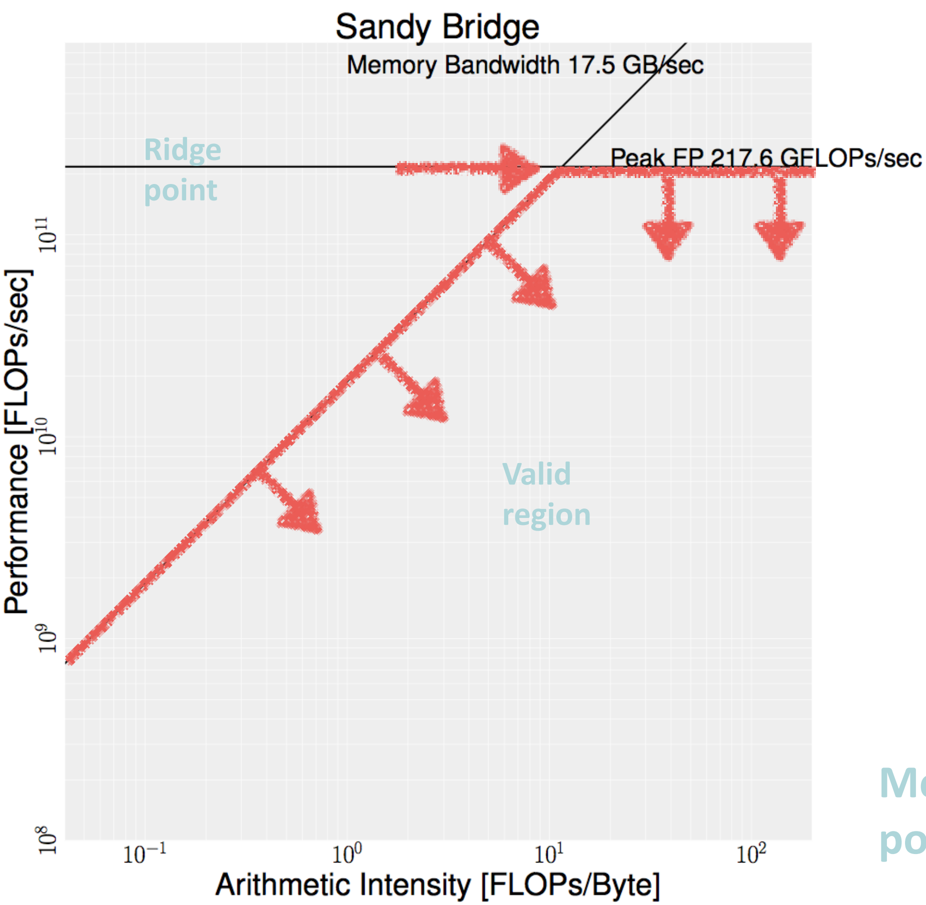


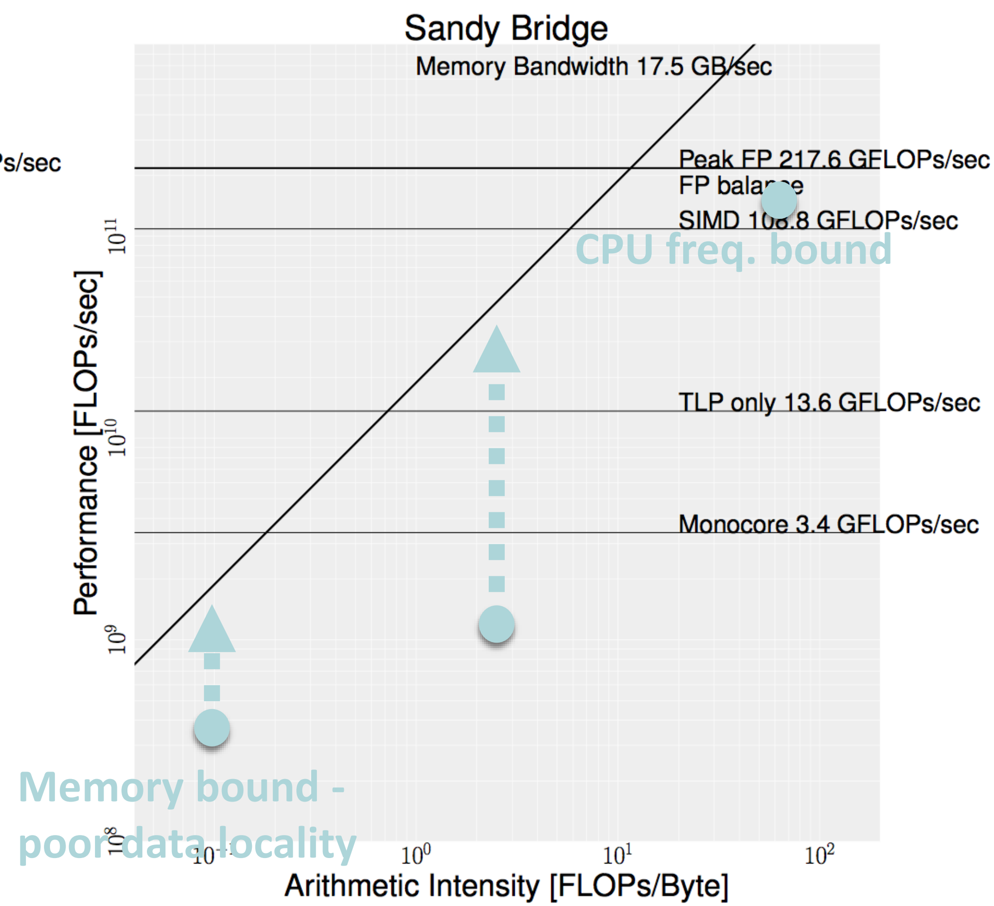
Hennessy and Patterson's Computer Architecture (5th ed.)

* E5-2690 v3 aka Haswell (launched 2014) ** Kepler (2013)

Roofline Model: Visual Performance Model

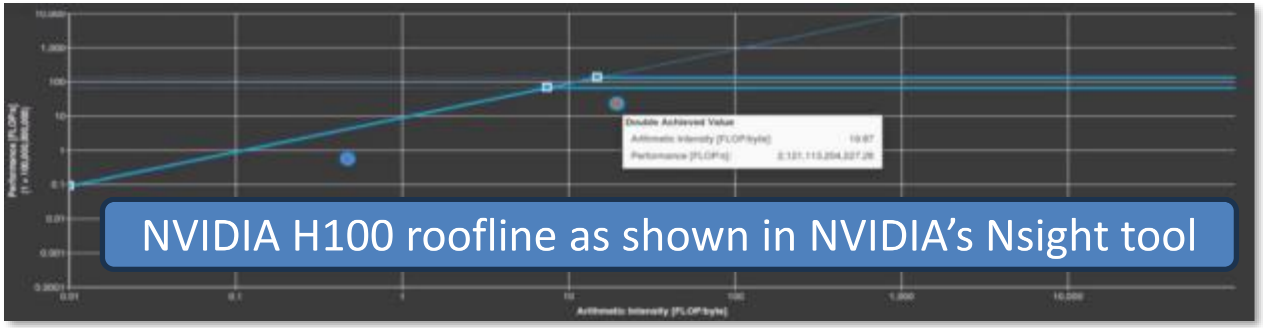
- Bound and bottleneck analysis (like Amdahl's law)
- Relates processor performance to off-chip memory traffic (bandwidth often the bottleneck)





Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures, Samuel Williams et al, CACM 2008

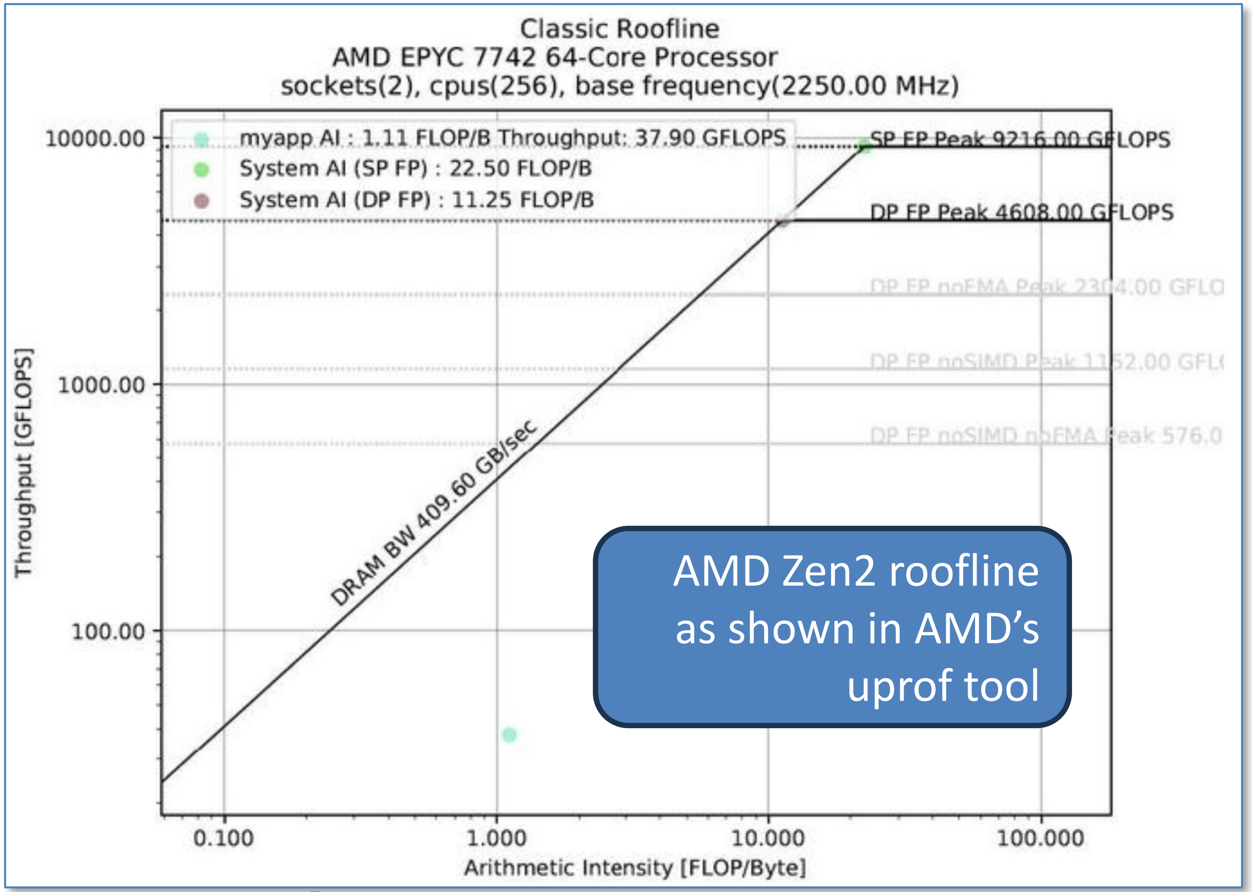
Roofline Model: Visual Performance Model



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

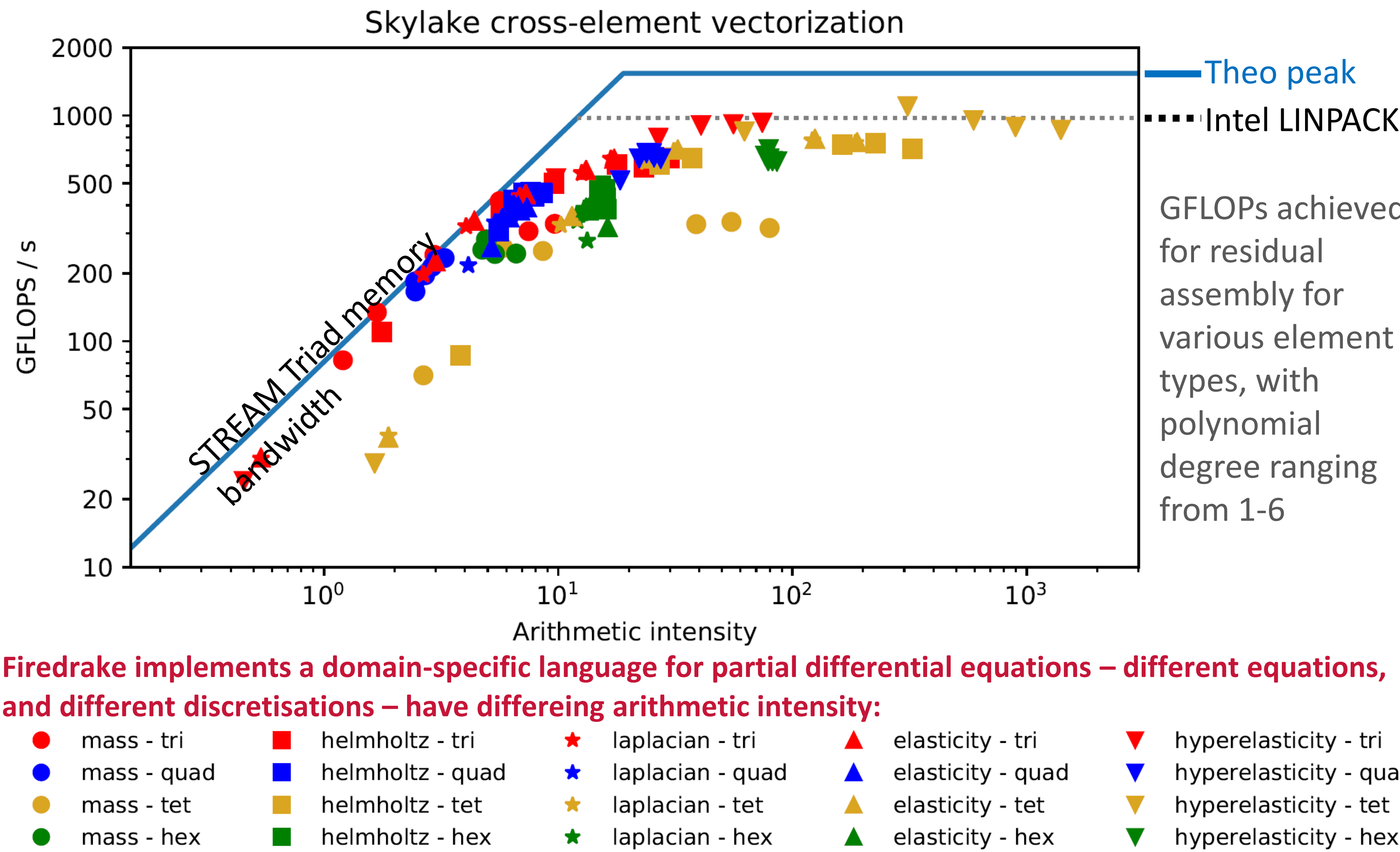
- The "roofline" concept is often used in performance optimisation tools
- To show how close you are to the limit
- And what you can do about it

- 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=tGwsnZYhKNayV9CClRa18Q

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



GFLOPs achieved for residual assembly for various element types, with polynomial degree ranging from 1-6

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

[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

More formally…

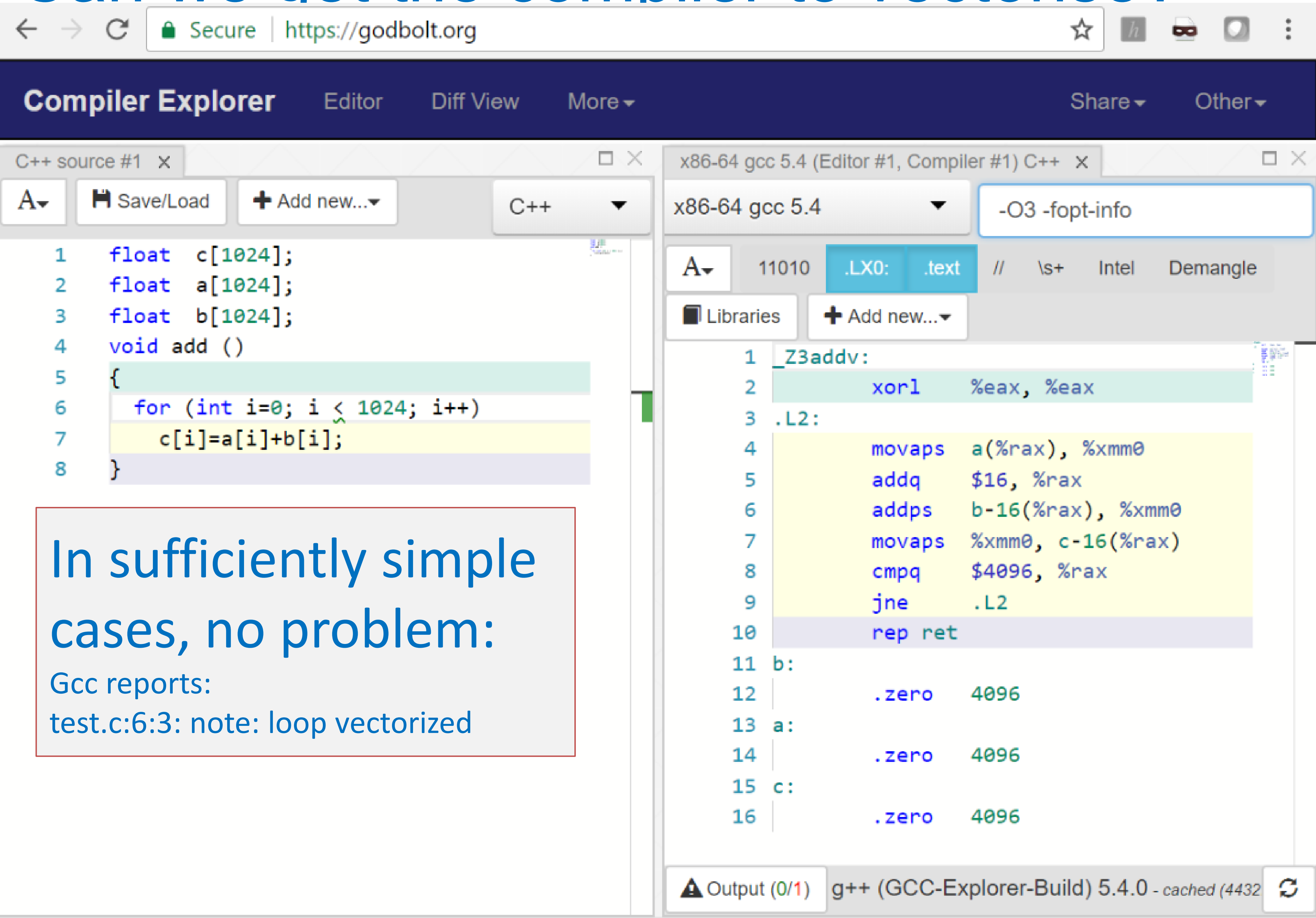
AVX512: vector addition

- 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

```
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

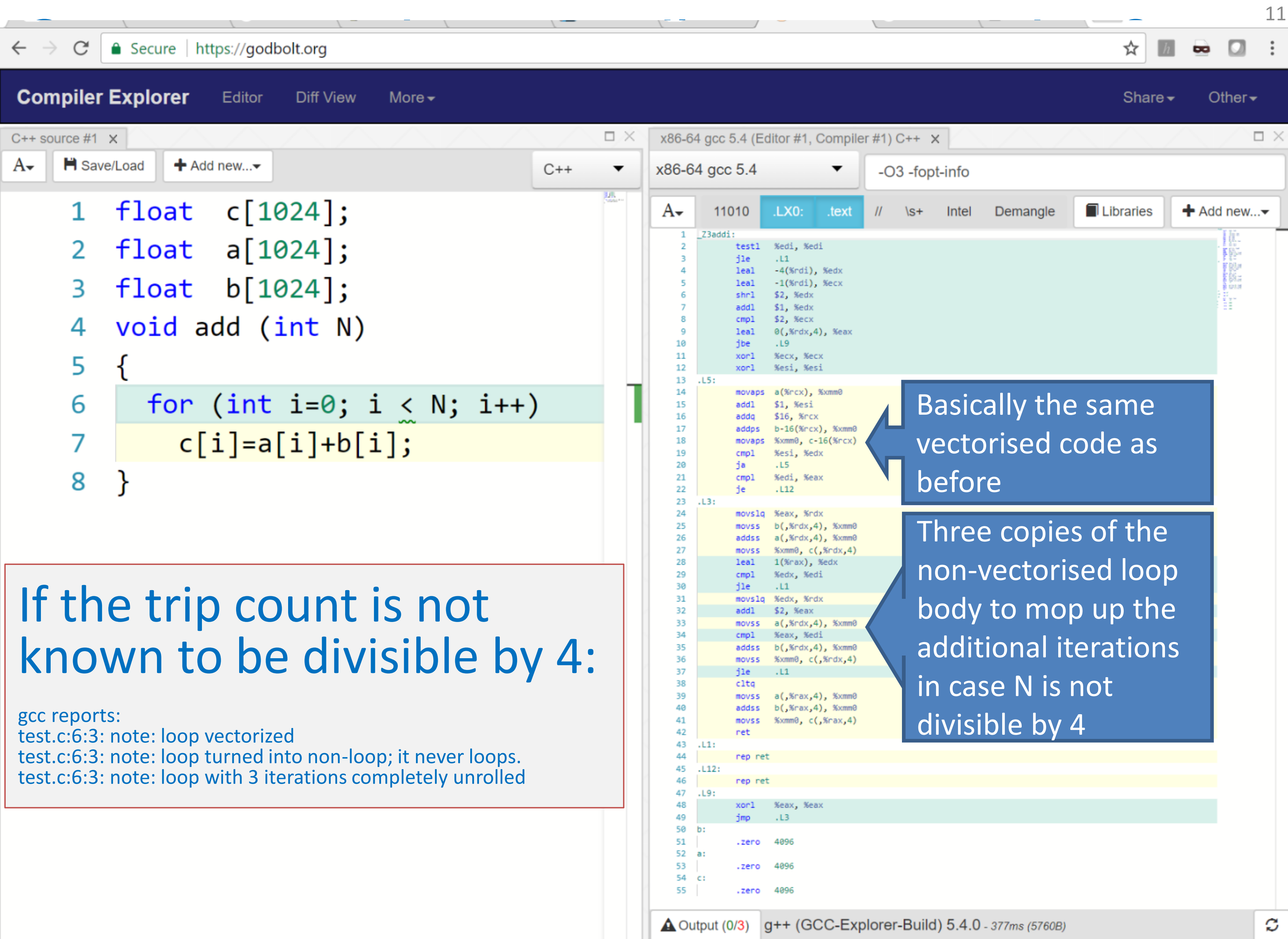
Can we get the compiler to vectorise?



In sufficiently simple cases, no problem:

Gcc reports:

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



Basically the same vectorised code as before

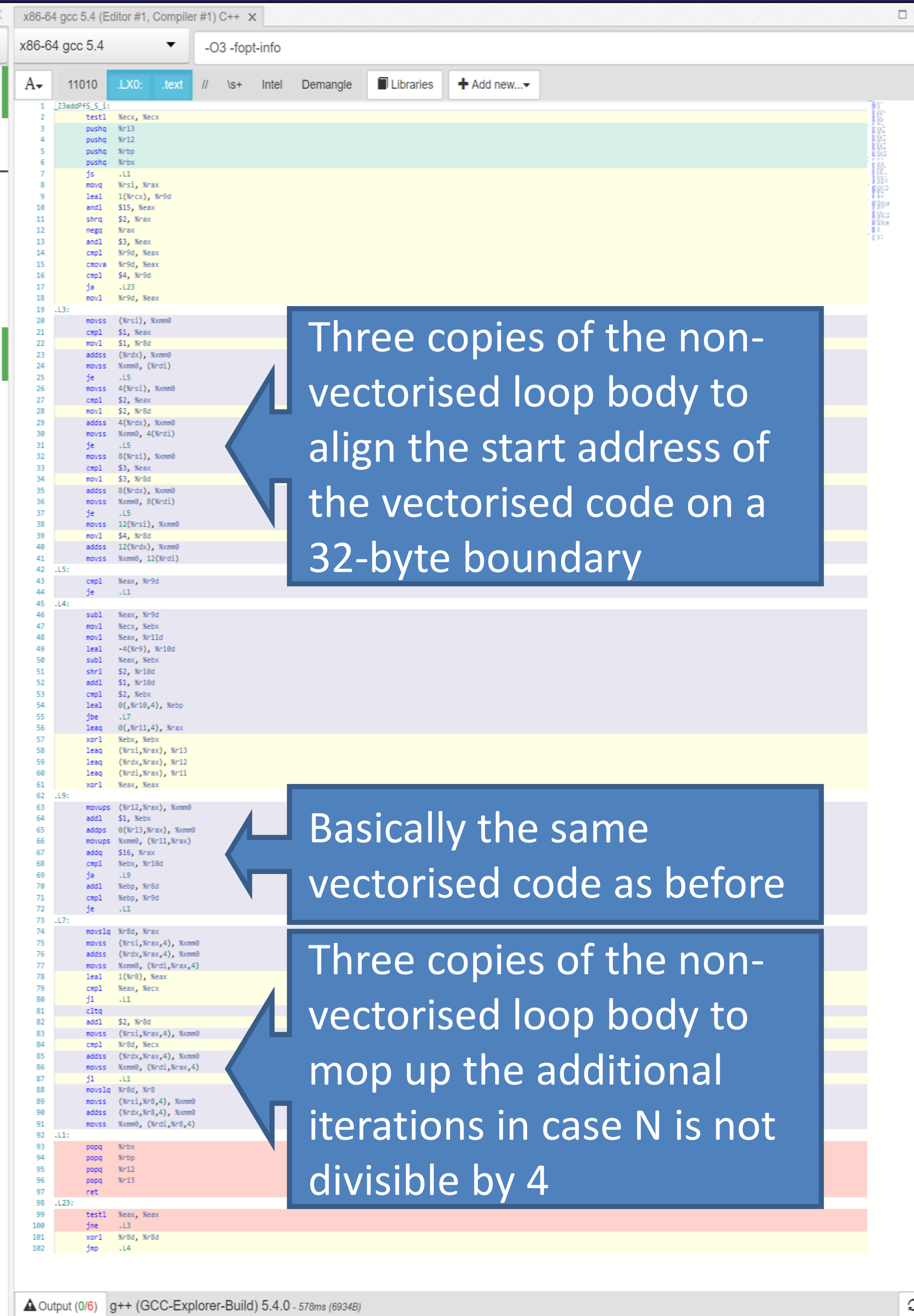
Three copies of the non-vectorised loop body to mop up the additional iterations in case N is not divisible by 4

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

C++ source #1

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



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:

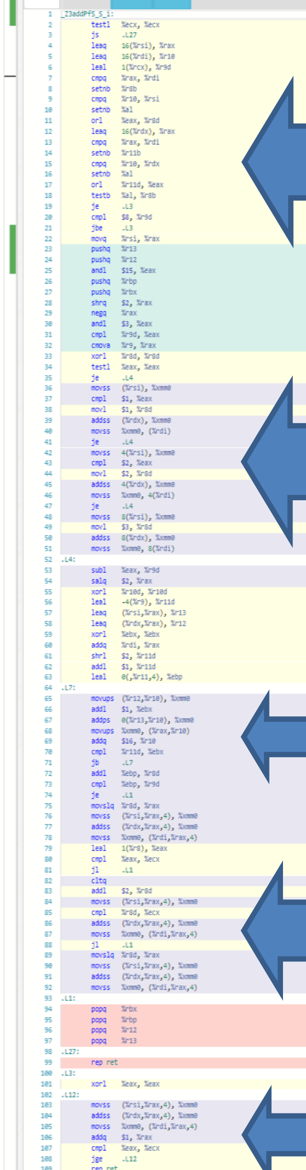
```
#include <immintrin.h> // For SSE/AVX instructions

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

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

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

x86-64 gcc 5.4 -O3 -fopt-info



Check whether the memory regions pointed to by c, b and a might overlap

Three copies of the non-vectorised loop body to align the start address of the vectorised code on a 32-byte boundary

Basically the same vectorised code as before

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

Non-vector version of the loop for the case when c might overlap with a or b

If the pointers might be aliases:

gcc reports:

test.c:6:3: note: loop vectorized
test.c:6:3: note: loop versioned for vectorization because of possible aliasing
test.c:6:3: note: loop peeled for vectorization to enhance alignment
test.c:6:3: note: loop turned into non-loop; it never loops.
test.c:6:3: note: loop with 3 iterations completely unrolled
test.c:1:6: note: loop turned into non-loop; it never loops.
test.c:1:6: note: loop with 3 iterations completely unrolled

Output (0/7) g++ (GCC-Explorer-Build) 5.4.0 - 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…

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

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: SIMT

Basically… think of each lane as a thread

Or: vectorise an *outer* loop:

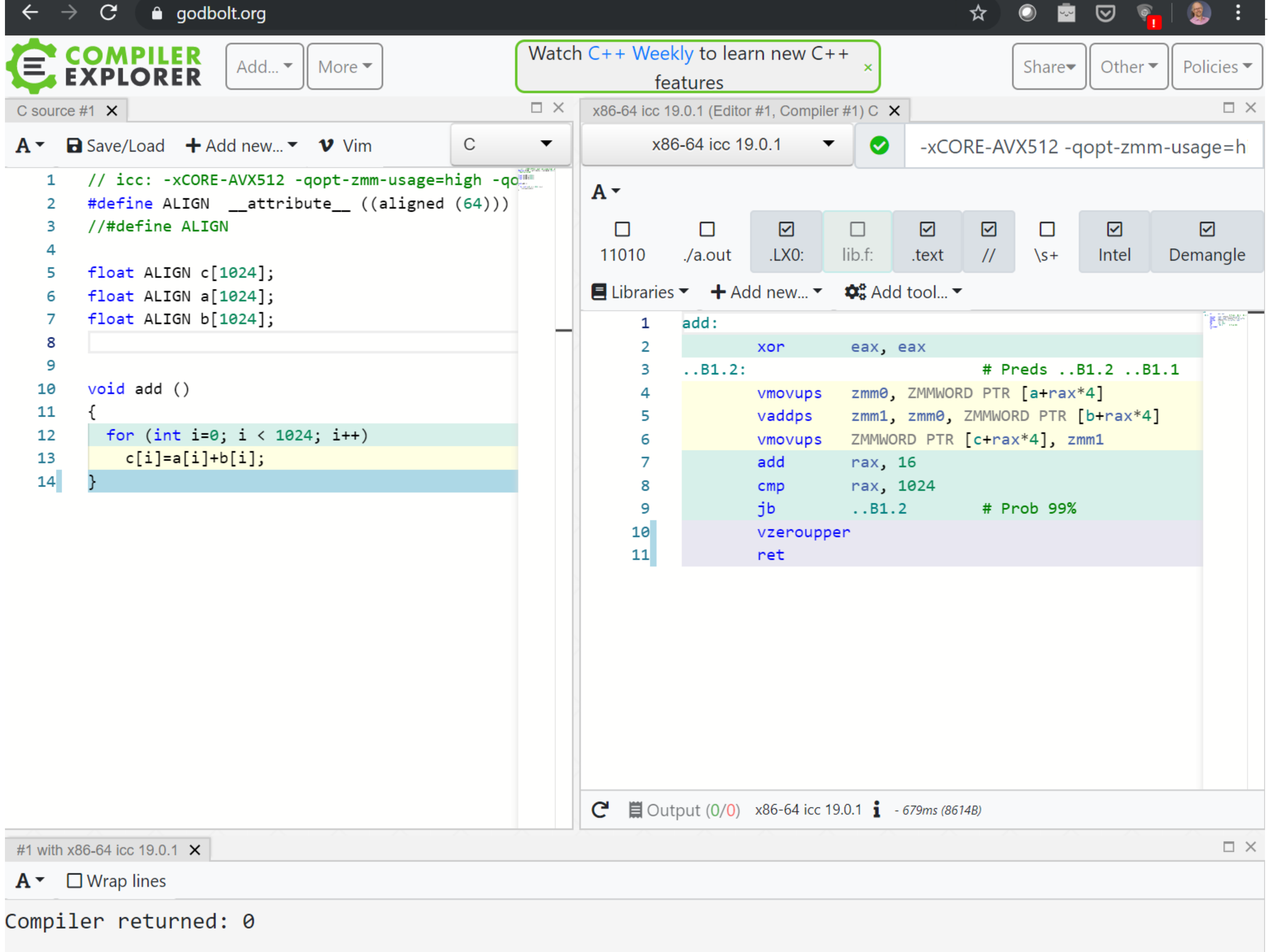
```
#pragma omp simd
for (int i=0; i<N; ++i) {
  if(){…} else {…}
  for (int j=….) {…}
  while(…) {…}
  f(…)
}
```

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

Use predication to handle:

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

More later – when we look at GPUs


godbolt.org
COMPILER EXPLORER
Add...
More
Watch C++ Weekly to learn new C++ features
Share
Other
Policies
C source #1
Save/Load
Add new...
Vim
C
// icc: -xCORE-AVX512 -qopt-zmm-usage=high -qo
#define ALIGN __attribute__ ((aligned (64)))
//#define ALIGN
float ALIGN c[1024];
float ALIGN a[1024];
float ALIGN b[1024];
void add ()
{
for (int i=0; i < 1024; i++)
c[i]=a[i]+b[i];
}
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...
add:
xor eax, eax
..B1.2: # Preds ..B1.2 ..B1.1
vmovups zmm0, ZMMWORD PTR [a+rax*4]
vaddps zmm1, zmm0, ZMMWORD PTR [b+rax*4]
vmovups ZMMWORD PTR [c+rax*4], zmm1
add rax, 16
cmp rax, 1024
jb ..B1.2 # Prob 99%
vzeroupper
ret
Output (0/0)
x86-64 icc 19.0.1
- 679ms (8614B)
#1 with x86-64 icc 19.0.1
Wrap lines
Compiler returned: 0

godbolt.org

COMPILER EXPLORER

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

C source #1

```c
// icc: -xCORE-AVX512 -qopt-zmm-usage=high -qo
#define ALIGN  __attribute__ ((aligned (64)))
//#define ALIGN

float ALIGN c[1024];
float ALIGN a[1024];
float ALIGN b[1024];
int ALIGN ind[1024];

void add ()
{
  for (int i=0; i < 1024; i++)
    c[i]=a[i]+b[ind[i]];
}
```

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

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

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

Output (0/0) x86-64 icc 19.0.1 - 946ms (9359B)

#1 with x86-64 icc 19.0.1

```
Compiler returned: 0
```

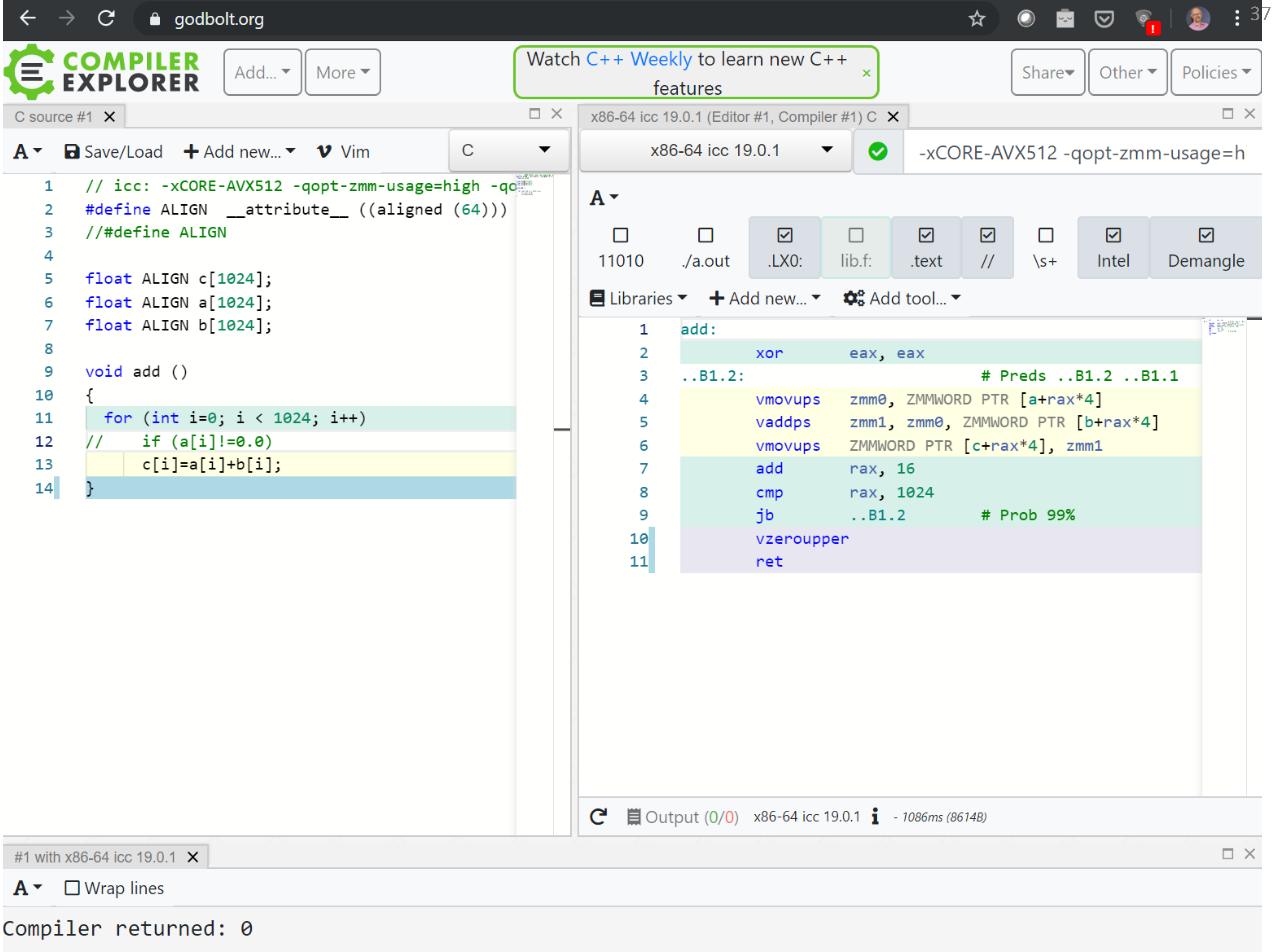
Indirection: b[ind[]]

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??)


godbolt.org
COMPILER EXPLORER
Add...
More
Watch C++ Weekly to learn new C++ features
Share
Other
Policies
C source #1
Save/Load
Add new...
Vim
C
// icc: -xCORE-AVX512 -qopt-zmm-usage=high -qo
#define ALIGN __attribute__ ((aligned (64)))
//#define ALIGN
float ALIGN c[1024];
float ALIGN a[1024];
float ALIGN b[1024];
void add ()
{
for (int i=0; i < 1024; i++)
// if (a[i]!=0.0)
c[i]=a[i]+b[i];
}
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...
add:
xor eax, eax
..B1.2: # Preds ..B1.2 ..B1.1
vmovups zmm0, ZMMWORD PTR [a+rax*4]
vaddps zmm1, zmm0, ZMMWORD PTR [b+rax*4]
vmovups ZMMWORD PTR [c+rax*4], zmm1
add rax, 16
cmp rax, 1024
jb ..B1.2 # Prob 99%
vzeroupper
ret
Output (0/0)
x86-64 icc 19.0.1
- 1086ms (8614B)
#1 with x86-64 icc 19.0.1
Wrap lines
Compiler returned: 0

godbolt.org

COMPILER EXPLORER

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

C source #1

```c
// icc: -xCORE-AVX512 -qopt-zmm-usage=high -qo
#define ALIGN   __attribute__ ((aligned (64)))
//#define ALIGN

float ALIGN c[1024];
float ALIGN a[1024];
float ALIGN b[1024];

void add ()
{
  for (int i=0; i < 1024; i++)
    if (a[i]!=0.0)
      c[i]=a[i]+b[i];
}
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C — -xCORE-AVX512 -qopt-zmm-usage=h

```asm
add:
        xor       eax, eax
        vpxord    zmm0, zmm0, zmm0
..B1.2:                         # Preds ..B1.2 ..B1.1
        vmovups   zmm1, ZMMWORD PTR [a+rax*4]
        vcmpps    k1, zmm1, zmm0, 4
        vaddps    zmm2, zmm1, ZMMWORD PTR [b+rax*4]
        vmovups   ZMMWORD PTR [c+rax*4]{k1}, zmm2
        add       rax, 16
        cmp       rax, 1024
        jb        ..B1.2        # Prob 99%
        vzeroupper
        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

#1 with x86-64 icc 19.0.1

```
Compiler returned: 0
```

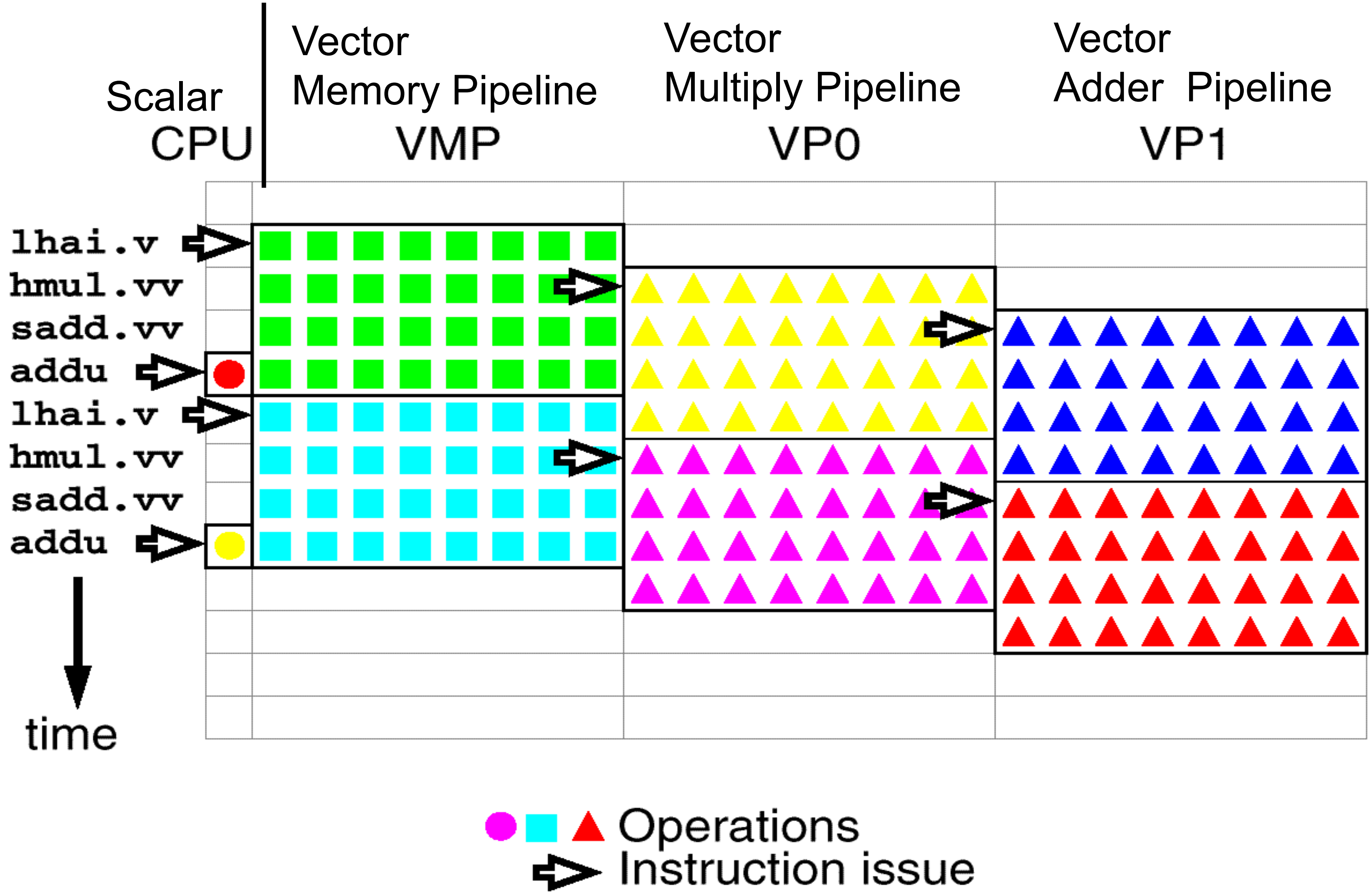
Vector execution alternatives

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**

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**

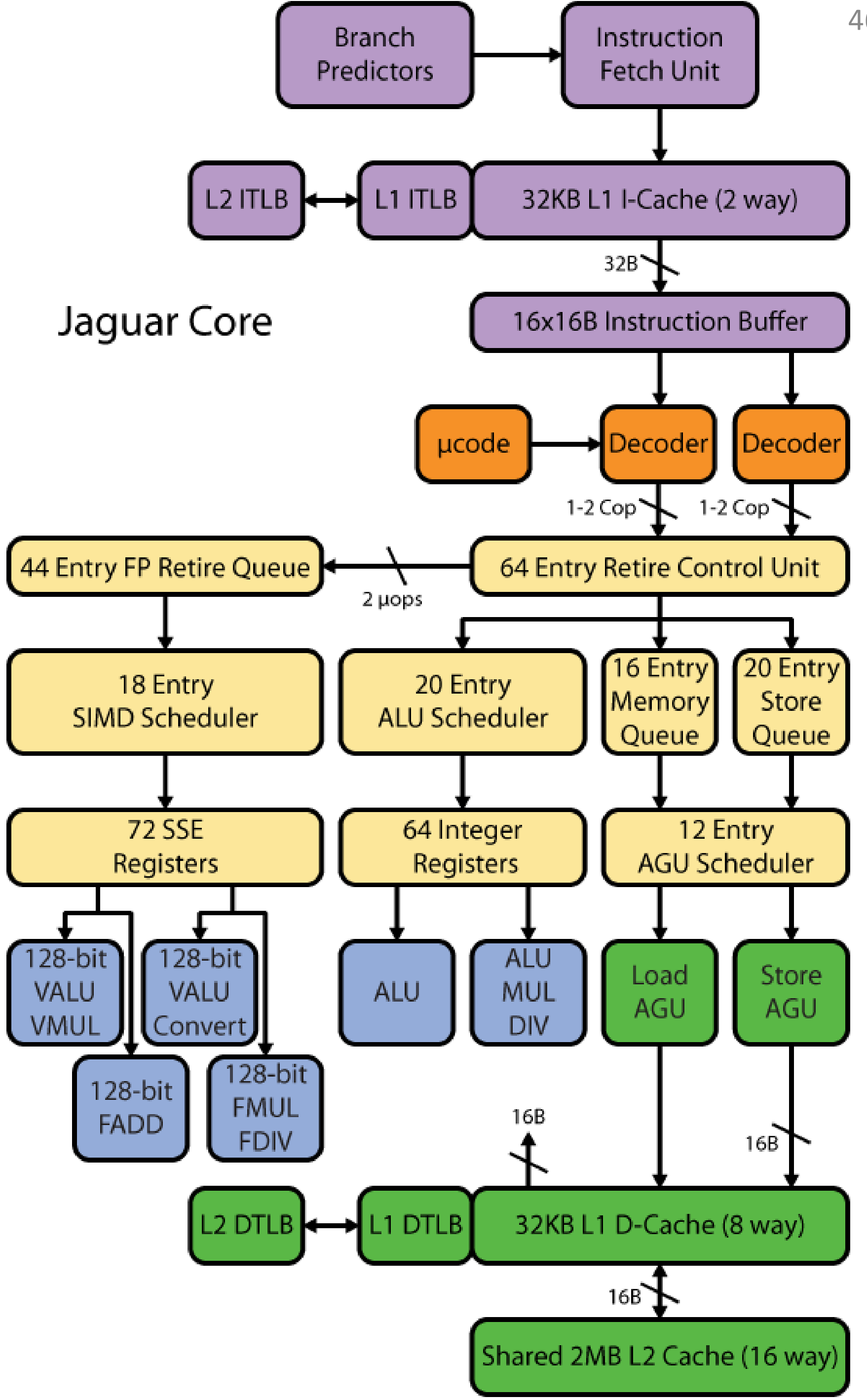


Figure from http://www.realworldtech.com/jaguar/4/ which includes more detail

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, 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, 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).

- AV512 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".

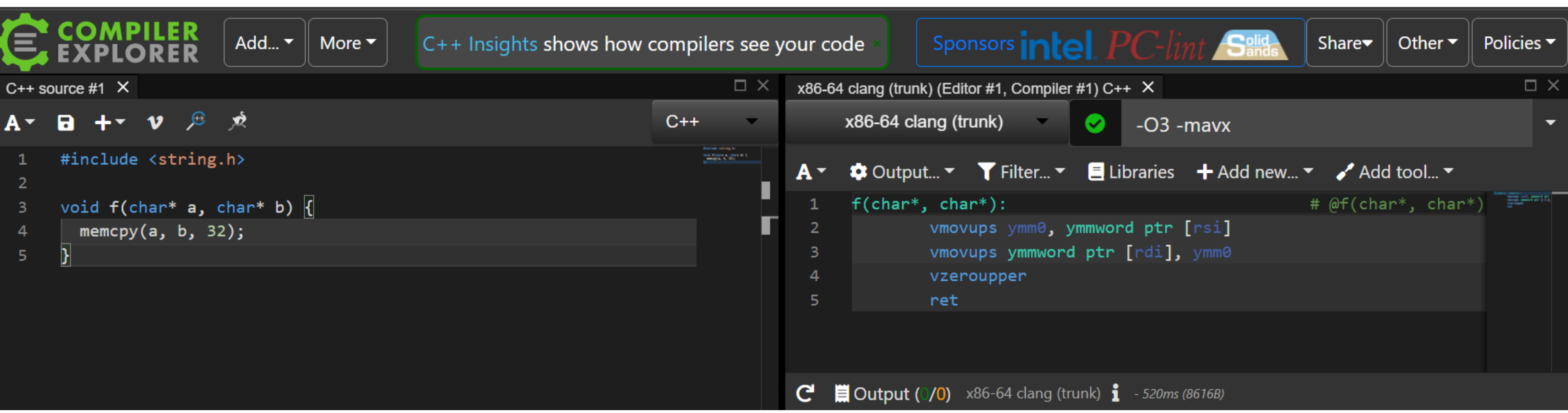
- SIMT to SIMD translation gets tricky if the loop body contains an if-then. For this, AVX512 uses the idea of "predication". For this purpose it has one-bit-per-lane predicate registers k0-k7. These registers can be used to control which lanes of a vector instruction are active and which lanes do nothing.

Summary Vectorisation Solutions

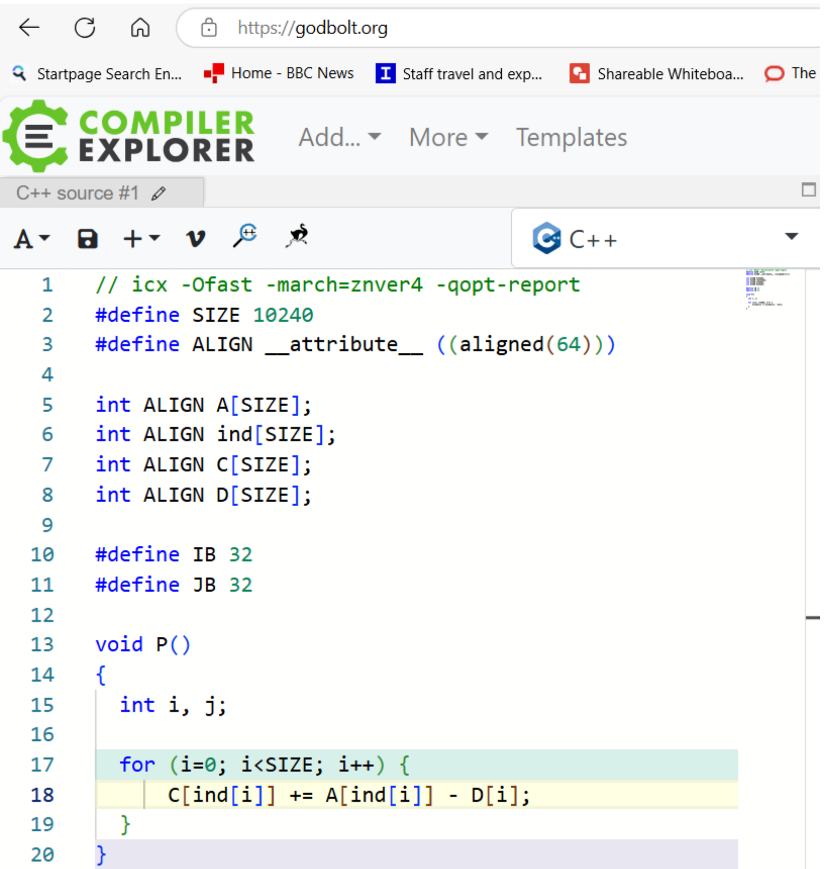
1. Indirectly through high-level libraries/code generators
2. Auto-vectorisation (eg use "-O3 –mavx2 –fopt-info" and hope it vectorises):
   - code complexity, sequential languages and practices get in the way
   - Give your compiler hints and hope it vectorises:
     - C99 "restrict" (implied in FORTRAN 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?

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

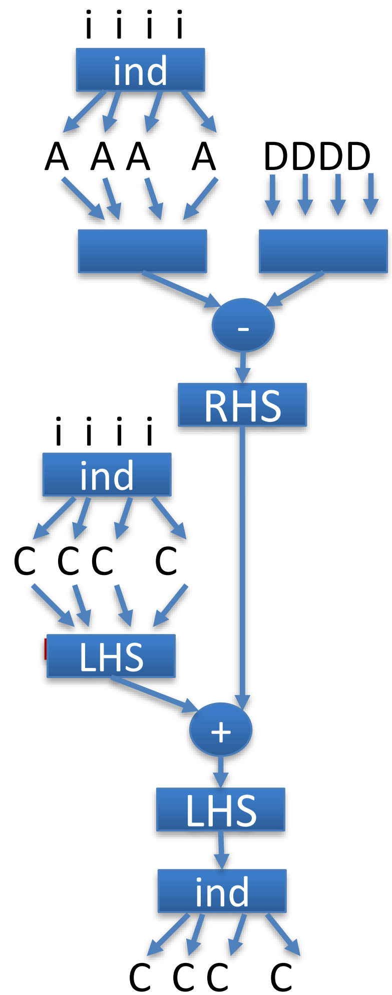


More things in case you're interested…

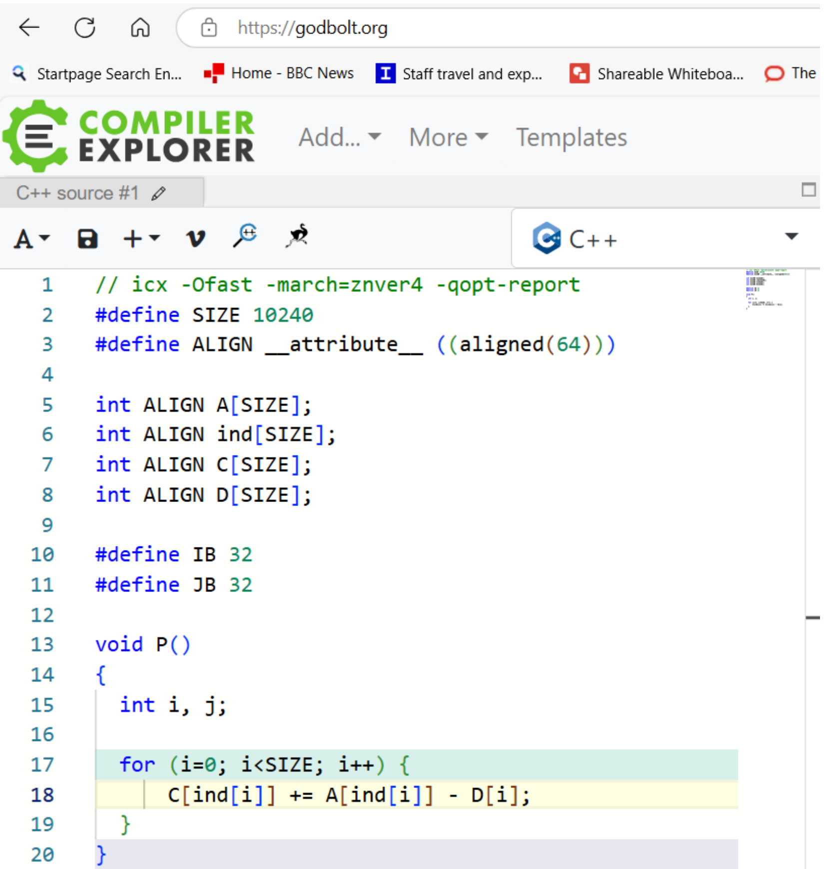


Incrementing through indirection: ind[i]

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

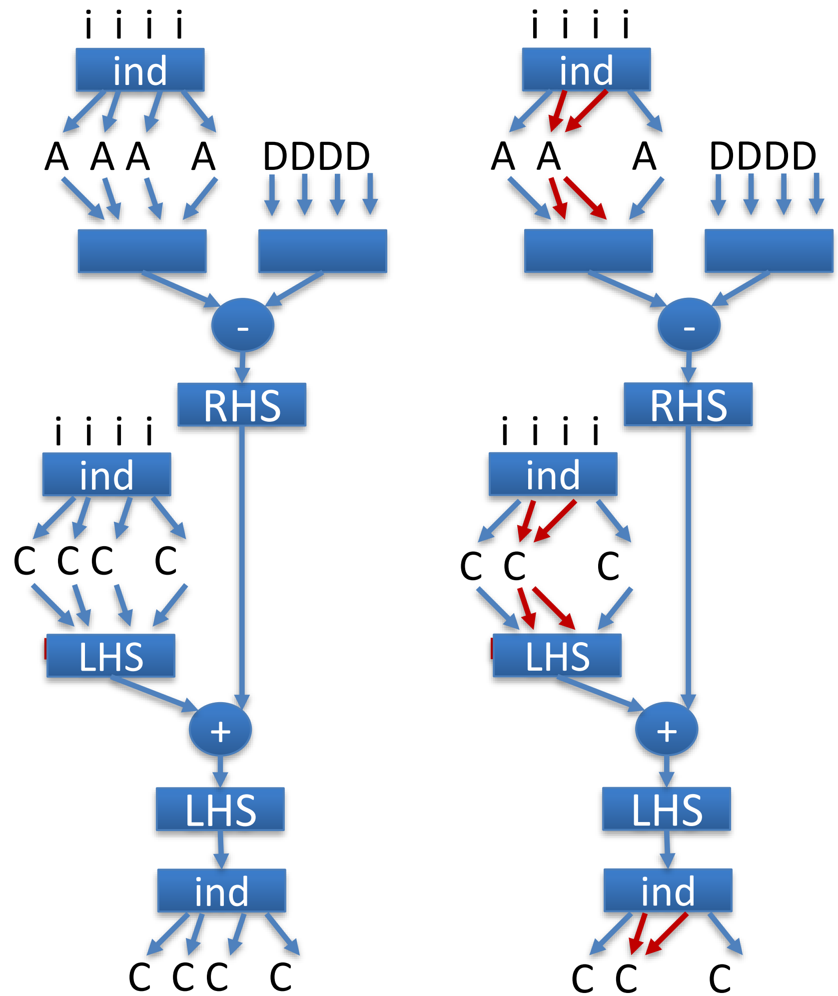


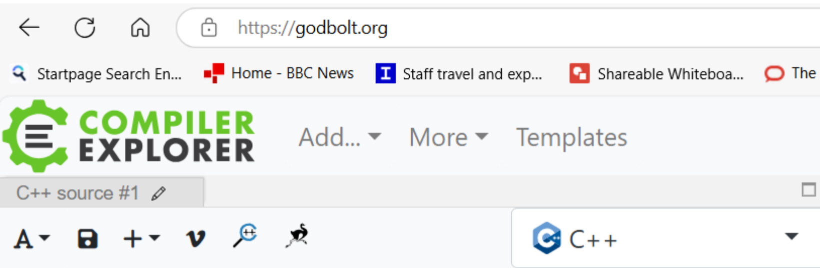
What would happen if there were duplicate indices in ind?



Incrementing through indirection: ind[i]

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



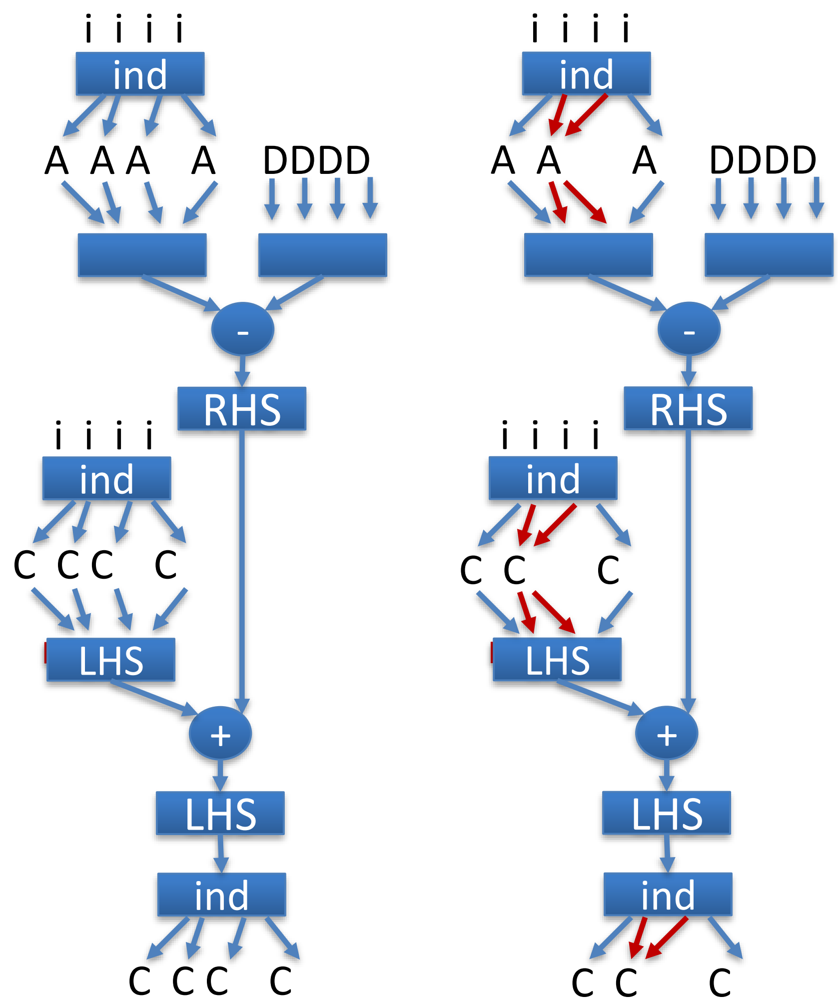



```
// icx -Ofast -march=znver4 -qopt-report
#define SIZE 10240
#define ALIGN __attribute__ ((aligned(64)))

int ALIGN A[SIZE];
int ALIGN ind[SIZE];
int ALIGN C[SIZE];
int ALIGN D[SIZE];

#define IB 32
#define JB 32

void P()
{
  int i, j;

  for (i=0; i<SIZE; i++) {
    C[ind[i]] += A[ind[i]] - D[i];
  }
}
```

Incrementing through indirection: ind[i]

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

What would happen if there were duplicate indices in ind?



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

https://godbolt.org

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

COMPILER EXPLORER Add... More Templates

C++ source #1 C++

```cpp
// icx -Ofast -march=znver4 -qopt-report
#define SIZE 10240
#define ALIGN __attribute__ ((aligned(64)))

int ALIGN A[SIZE];
int ALIGN ind[SIZE];
int ALIGN C[SIZE];
int ALIGN D[SIZE];

#define IB 32
#define JB 32

void P()
{
  int i, j;

  for (i=0; i<SIZE; i++) {
      C[ind[i]] += A[ind[i]] - D[i];
  }
}
```

Incrementing through indirection: ind[i]

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

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

Add RHS into LHS

RHS

Count conflicts

If no conflicts the skip to fast case

Add conflicting lanes' values sequentially

Unrolled copy

Add RHS into LHS

Scatter back

Start next iteration

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

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

Roughly…

Not examinable

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

Health warning

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

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

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

Example 4:

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

```
for i = 1 to N
  a[3*i] = a[5*1-20] + b[i]
Dependence is found, for example when:
i1 = 5 (LHS)
i2 = 7 (RHS)
Loop-carried true dependence found, for example when:
i1 = 5
i2 = 7
Loop-carried anti-dependence found, for example when:
i1 = 15
i2 = 13
```

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

S1 : A[0] := 0

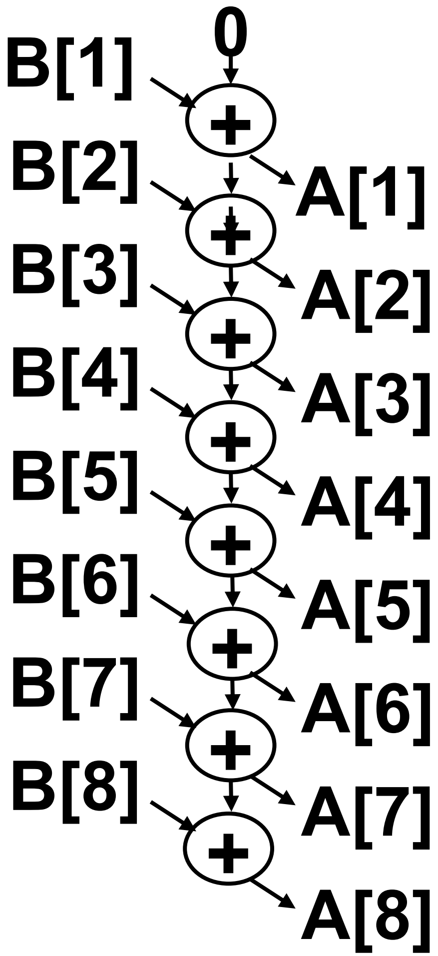
for i = 1 to 8

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

Feeding curiosity (not examinable)

Loop-carried dependences can sometimes still be parallelised

- **Appears to be inherently sequential**

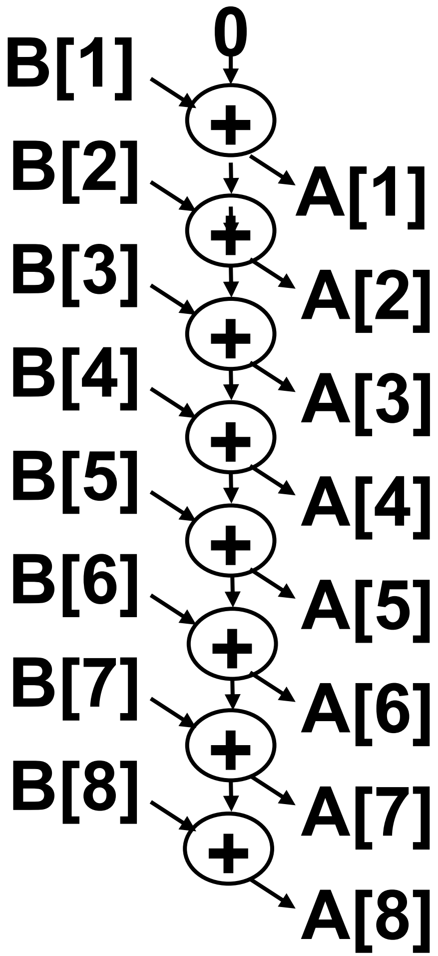


```
S1 :  A[0] := 0
      for i = 1 to 8
S2 :    A[i] := A[i-1] + B[i]
```

Feeding curiosity (not examinable)

Loop-carried dependences can sometimes still be parallelised

- **Appears to be inherently sequential**



- **But parallel is possible:**

B:	1	1	1	1	1	1	1	1	
>>1:		1	1	1	1	1	1	1	**+**
A1:	1	2	2	2	2	2	2	2	
>>2:			1	2	2	2	2	2	**+**
A2:	1	2	3	4	4	4	4	4	
>>4:					1	2	3	4	**+**
A3:	1	2	3	4	5	6	7	8	

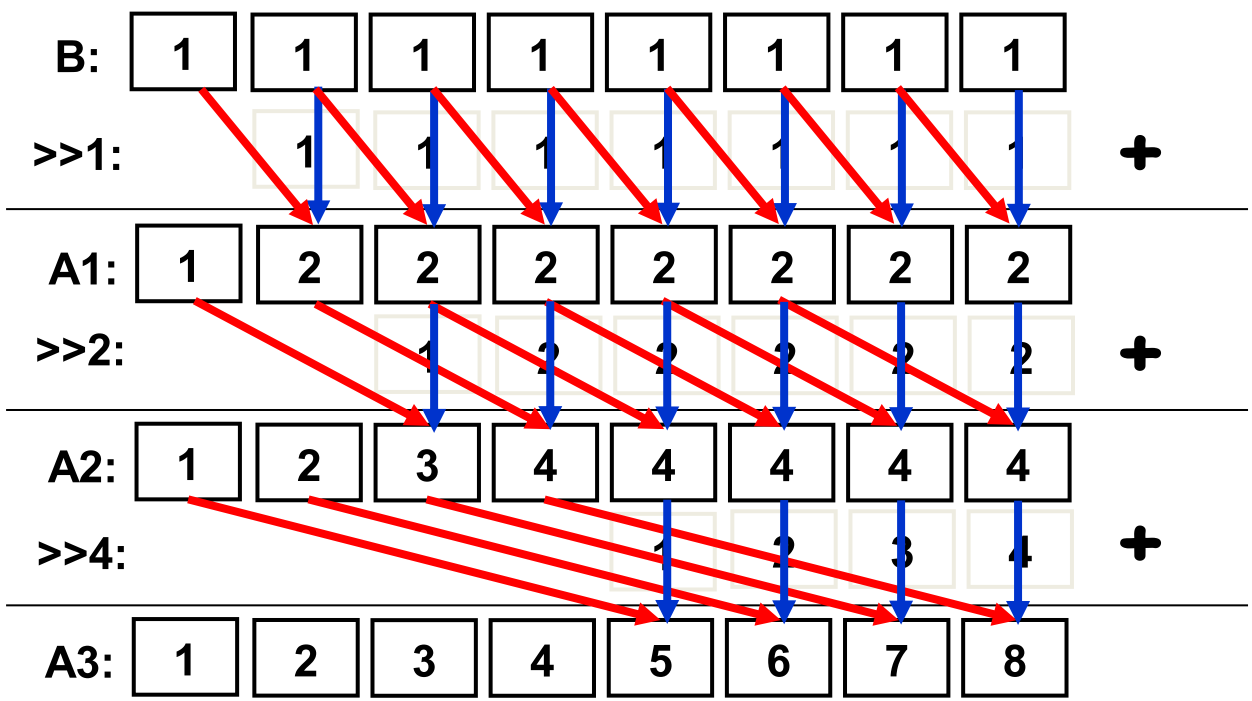
“Parallel scan” or “parallel prefix sum”

Feeding curiosity (not examinable)

Loop-carried dependences can sometimes still be parallelised

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

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



"Parallel scan" or "parallel prefix sum"

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

S1 : A[0] := 0

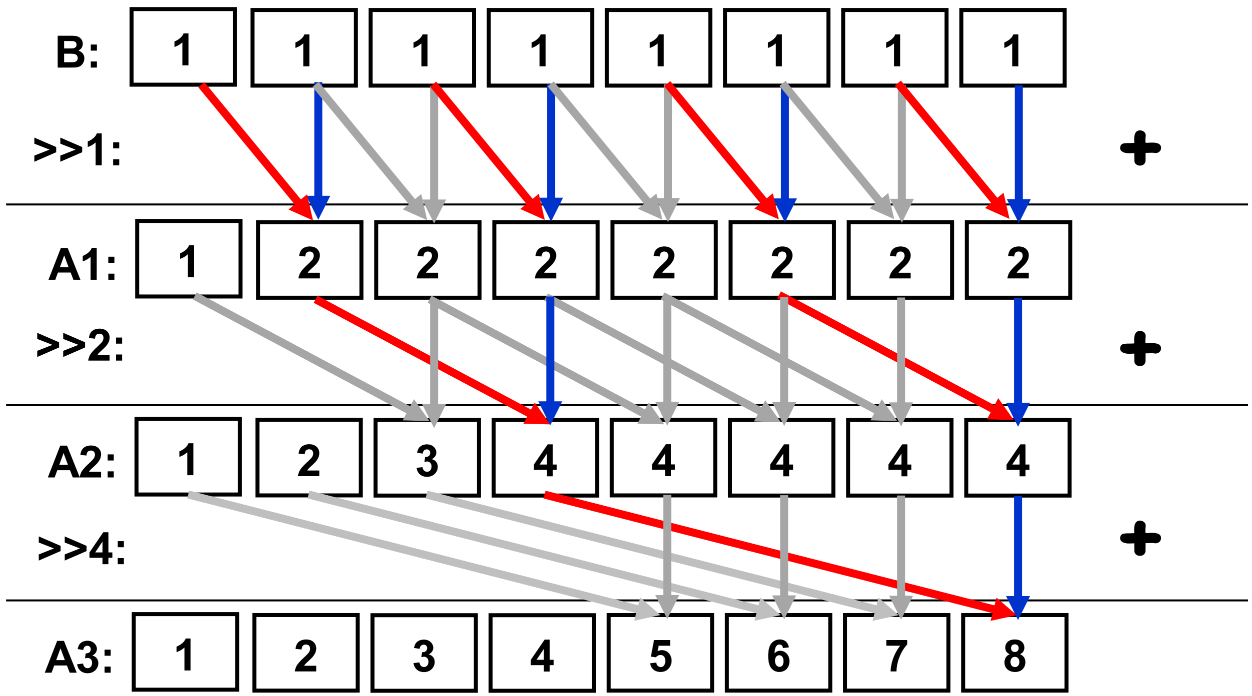
for i = 1 to 8

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**



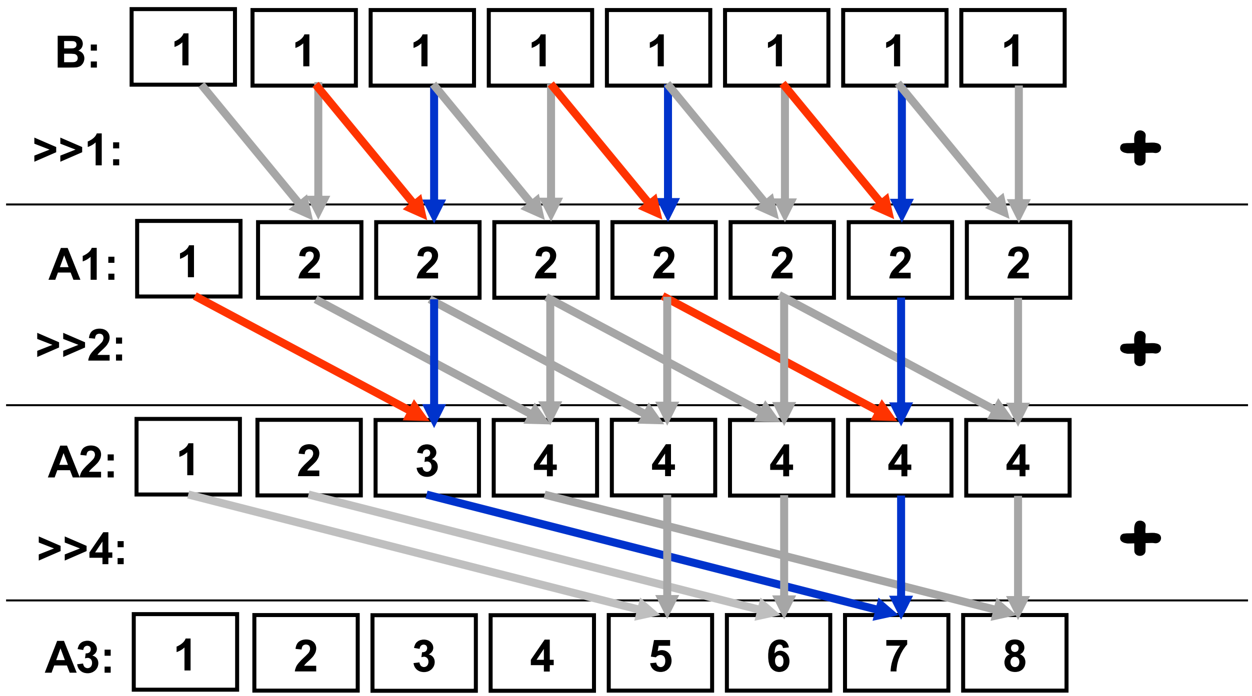
We can see that the last element is computed with a reduction tree

```
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

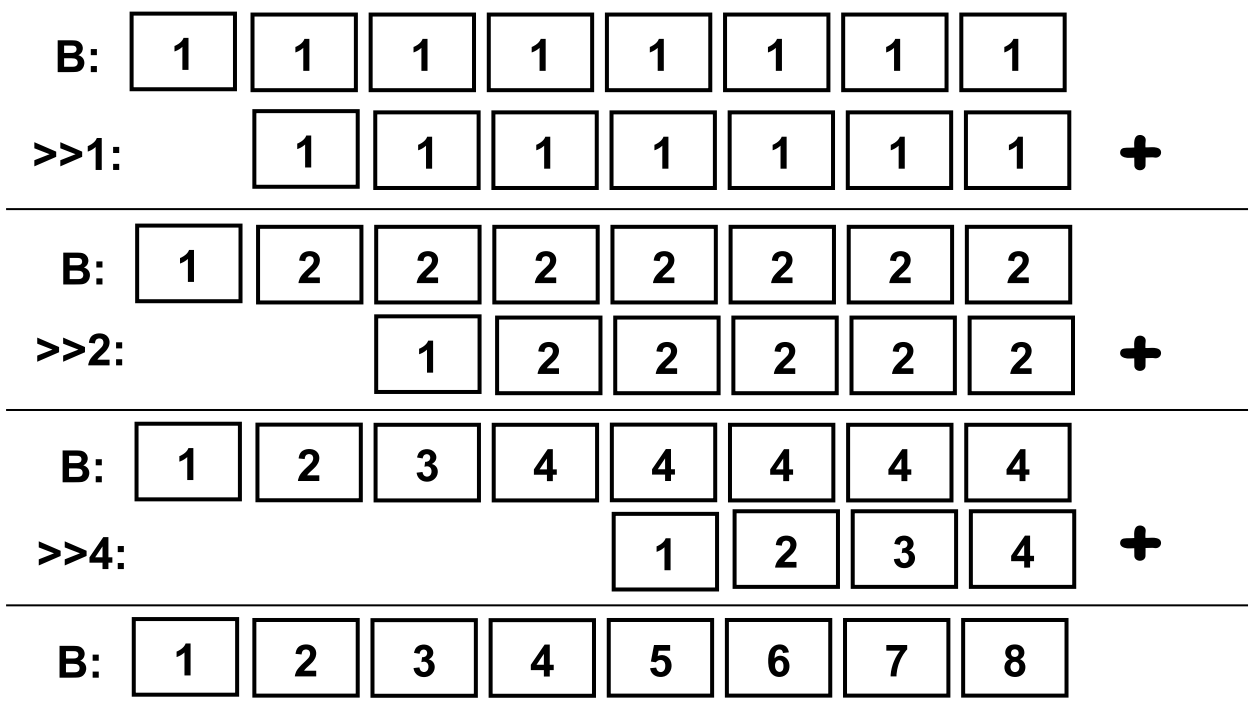
Feeding curiosity

S1 : A[0] := 0

for i = 1 to 8

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

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



“Parallel scan” or “parallel prefix sum”

- **This is the “naïve” parallel scan**
- **It does more work than the sequential scan – but it does use parallelism**
- **There are “work-efficient” parallel scans**
- **Eg see Mark Harris, GPU Gems Ch39** **https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda**