OpenMP for SIMD vectorization

N. Shukla

High-Performance Computing Department CINECA

Casalecchio di Reno Bologna, Italy

Email: n.shukla@cineca.it

www.cineca.it || Bologna 2022

Objectives

What is SIMD?

- Evolution of hardware (intel)
- Vector hardware
- Programming concept

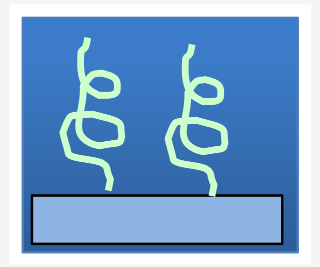
How to Program: OpenMP?

- SIMD loop directives
- SIMD Enabled function

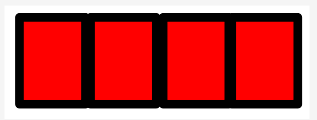
SIMD+Threads

- OpenMP

Hardware Diversity: Basic Building Blocks



CPU Core: one or more hardware threads sharing an address space



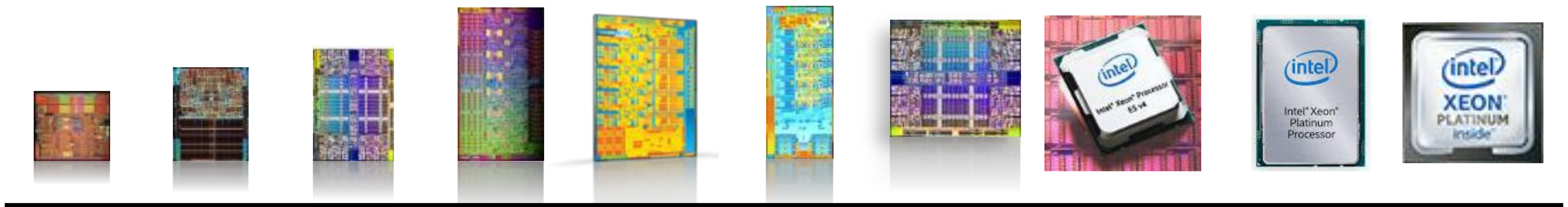
SIMD: Single Instruction Multiple Data
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.

Changing Hardware Impacts Software

More Cores→More Threads→Wider Vectors

High performance software must be both

- Parallel (multi-thread, multi-process)
- Vectorized

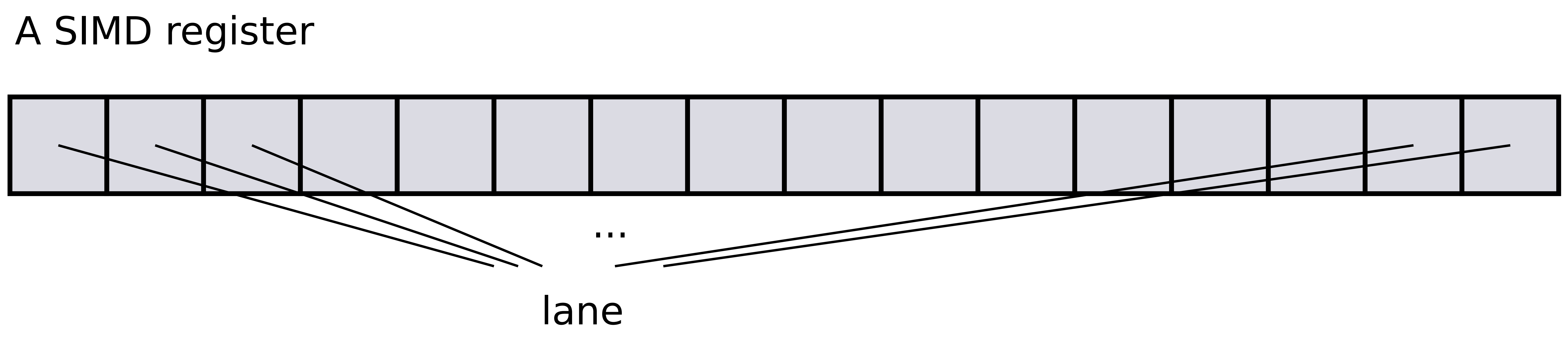


	Intel® Xeon® Processor								Intel® Xeon® Scalable Processor	
	64-bit	5100 series	5500 series	5600 series	E5-2600	E5-2600 V2	E5-2600 V3	E5-2600 V4	Platinum 8180	Platinum 9282
Core(s)	1	2	4	6	8	12	18	22	28	56
Threads	2	2	8	12	16	24	36	44	56	112
SIMD Width	128	128	128	128	256	256	256	256	512	512

* ENCCS/Intel Workshop on OpenMP Software Tools, 1 June 2021

SIMD: basic concepts

- SIMD : single instruction multiple data
- A SIMD register (or a vector register) can hold many values (4 - 16 values or more) of a single type
- SIMD instructions are hardware instructions that modify the vector registers
- Each value in a SIMD register is called a SIMD lane or simply **a** lane
- SIMD instructions can operate on several (typically all) values on a SIMD register



Vectorisation is done via SIMD instructions

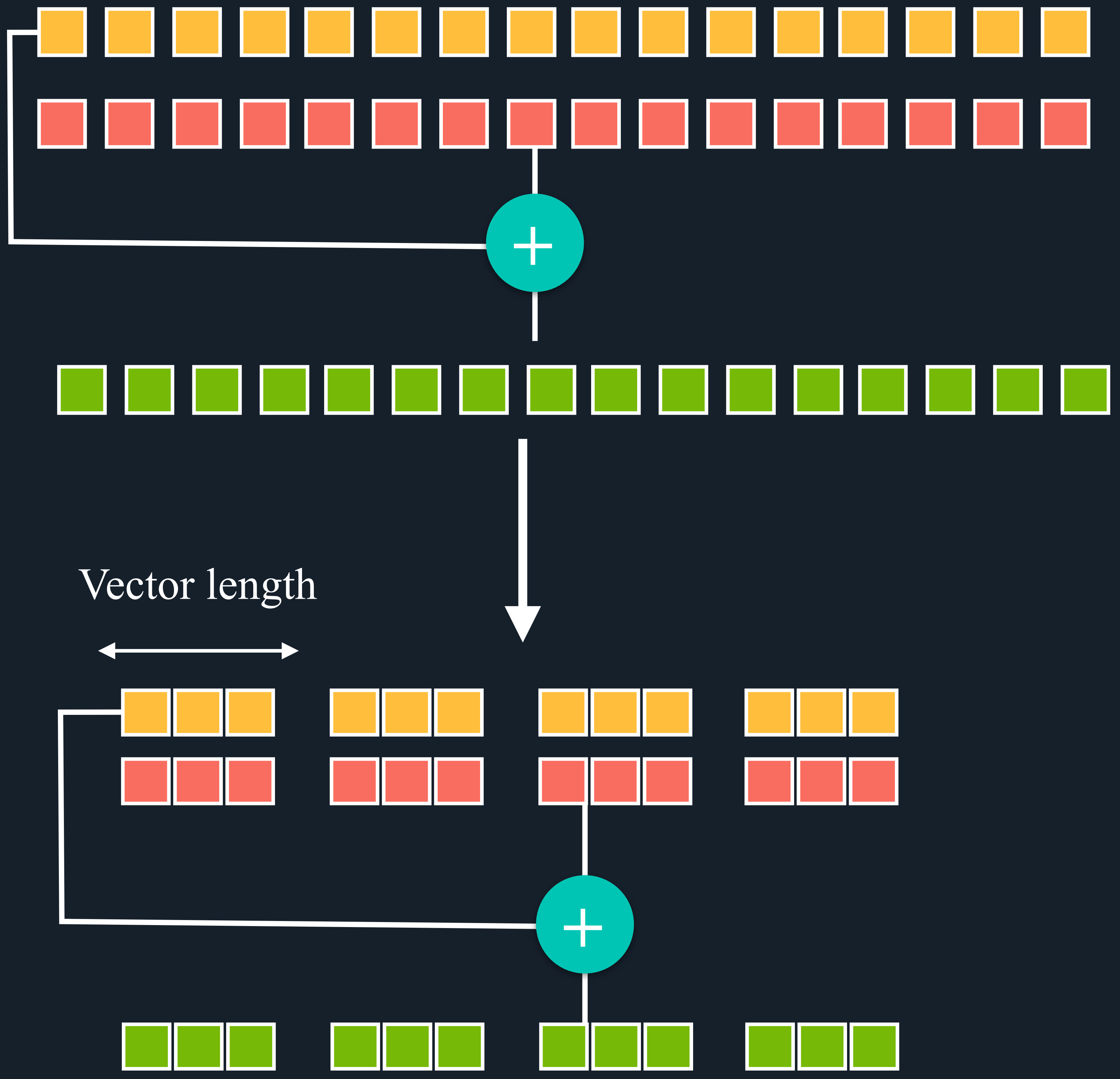
```
do i = 1, 16
    C[i] = A[i] + B[I]
end do
```

Scalar instructions

32 loads
16 adds
16 stores

SIMD instructions

8 loads
4 adds
4 stores



Vectorisation is referred as SIMD parallelism

Why vectorisation?

- Operates on entire blocks of data, called vector
- In OpenMP, vectorisation is referred to as SIMD parallelism
- It gives you more compute per cycle
- A single instruction operates upon multiple data elements concurrently
- Hence may increase the FLOP/s rate of the processor
- SIMD instruction use special SIMD registers containing multiple data elements
- Vectors help make good use of the memory hierarchy
- Vectorisation helps you write code which has good access patterns to **maximise bandwidth**

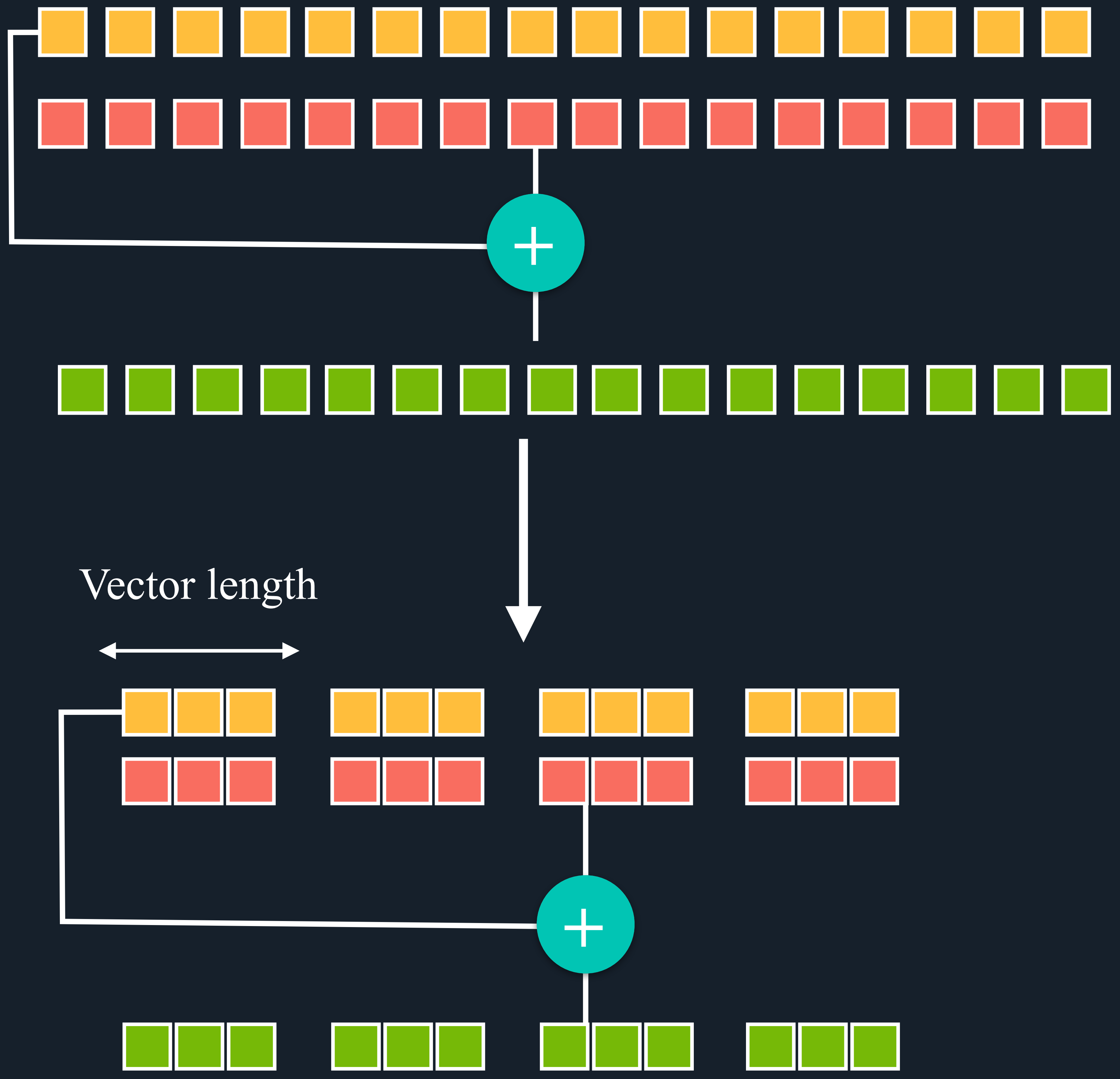
```
do i = 1, 16
   C[i] = A[i] + B[I]
end do
```

Scalar instructions

32 loads
16 adds
16 stores

SIMD instructions

8 loads
4 adds
4 stores



Explicit vectorization

Compiler responsibilities

- Allow programmer to declare that code can and should be run in SIMD
- Generate the code the programmer asked for

Programmer Responsibilities

- Correctness (e.g., no dependencies, no invalid memory accesses)
- Efficiency (e.g., alignment, loop order, masking)

Is your program vectorized ?

```
extern double *a;
extern double *b;
extern double *c;

for (int i = 0; i < 128; i++) {
  c[i] += a[i] * b[i];
}
```

```
vmovupd a+1024(%rcx), %ymm1
vmulpd  b+1024(%rcx), %ymm1, %ymm1
vaddpd  c+1024(%rcx), %ymm1, %ymm1
vmovupd %ymm1, c+1024(%rcx)
```

How to vectorize the code?

- Auto-vectorization

  Compiler can detect loops or blocks of codes that can be vectorized

  Auto-vectorization relies on static analysis

  Increased complexity of instructions makes it hard for the compiler to select proper instructions

  Code pattern needs to be recognized by the compiler

  Precision requirements often inhibit SIMD code gen

- Vectorization-report: Intel® Composer XE

  Intel:

  ```
  icc -qopenmp -qopt-report=N -qopt-report-phase=vec
  ```

  gnu:

  ```
  gcc -ftree-vectorize -ftree-vectorizer-verbose  automatically enabled with -O3
  ```

For details: https://gcc.gnu.org/projects/tree-ssa/vectorization.html

Compile report

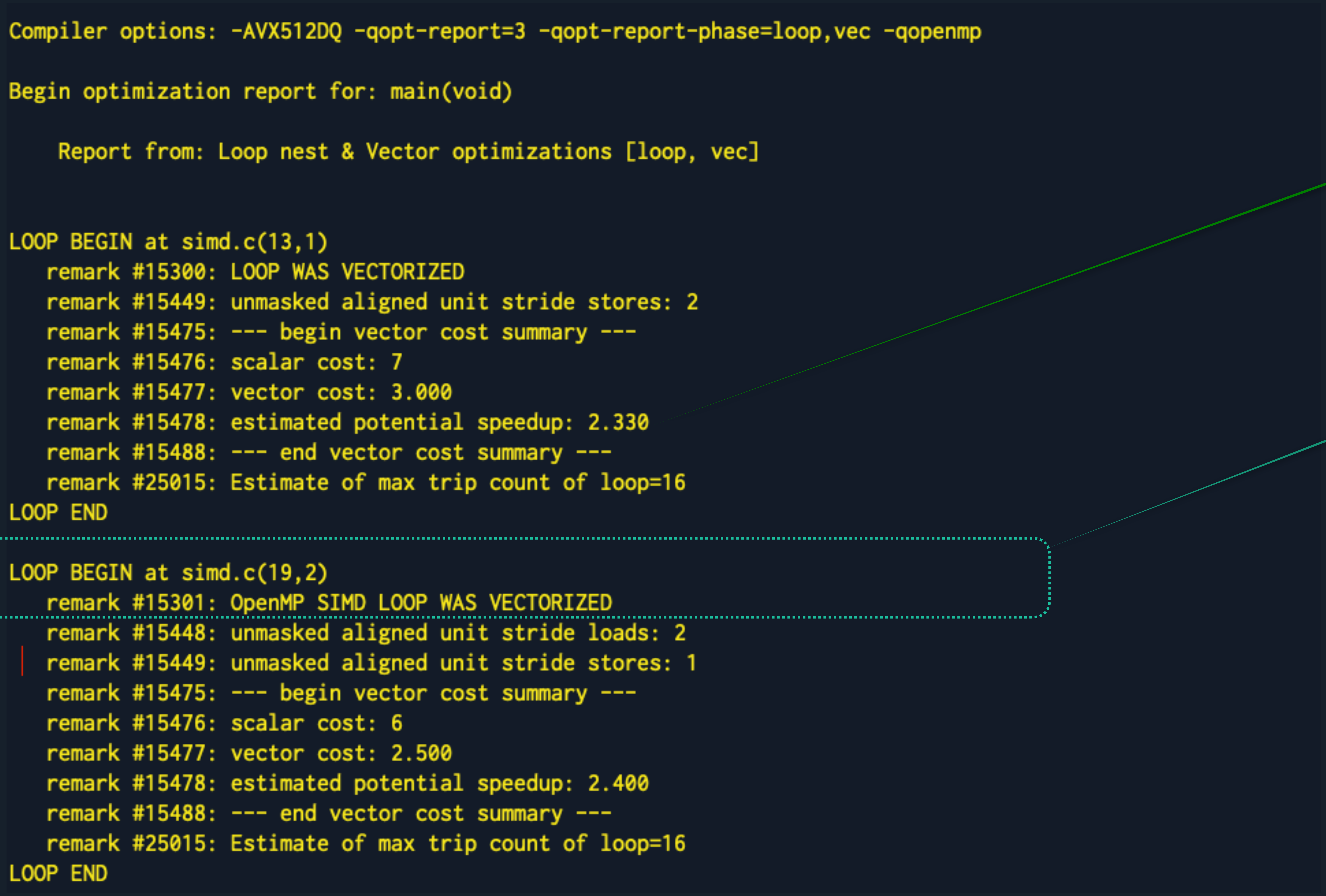
- -qopt-report[=n] : tells the compiler to generate an optimization report

  (Optional) Indicates the level of detail in the report. You can specify values 0 through 5. If you specify zero, no report is generated. For levels n=1 through n=5, each level includes all the information of the previous level, as well as potentially some additional information. Level 5 produces the greatest level of detail. If you do not specify n, the default is level 2, which produces a medium level of detail.
- -qopt-report-phase[=list]: specifies one or more optimizer phases for which optimization reports are generated

  (Optional) Indicates the level of detail in the report. You can specify values 0 through 5. If you specify zero, no report is generated. For levels n=1 through n=5, each level includes all the information of the previous level, as well as potentially some additional information. Level 5 produces the greatest level of detail. If you do not specify n, the default is level 2, which produces a medium level of detail
- -qopt-report-filter=string: specified the indicated parts of your application, and generate optimization reports for those parts of your application.

Optimization Report – An Example

```
$ icc -AVX512DQ -qopt-report=3 -qopt-report-phase=loop,vec -qopenmp simd.c
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
```



```
Compiler options: -AVX512DQ -qopt-report=3 -qopt-report-phase=loop,vec -qopenmp

Begin optimization report for: main(void)

    Report from: Loop nest & Vector optimizations [loop, vec]


LOOP BEGIN at simd.c(13,1)
   remark #15300: LOOP WAS VECTORIZED
   remark #15449: unmasked aligned unit stride stores: 2
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 7
   remark #15477: vector cost: 3.000
   remark #15478: estimated potential speedup: 2.330
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=16
LOOP END

LOOP BEGIN at simd.c(19,2)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 6
   remark #15477: vector cost: 2.500
   remark #15478: estimated potential speedup: 2.400
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=16
LOOP END
```

Potential speed up

OPENMP SIMD operation

Why auto-vectorization fails?

Data dependencies

Other potential reasons

- Alignement
- Function calls in the loop block
- Complex control flow/ conditional branches
- Loop not countable (E.g. upper bound not a runtime constant)
- Mixed data types
- Non-unit stride between elements
- Loop body too complex (register pressure)
- Vectorization seem inefficient

Many more … but less likely to occur

Modern compilers are very good at automatically vectorising the loops

Compiler needs to be sure it's safe to vectorise

Read compiler reports to see if it's already vectorising.

- Intel: -qopt-report=5
- Cray: -hlist=a
- GNU -fopt-info-vec

Data dependencies

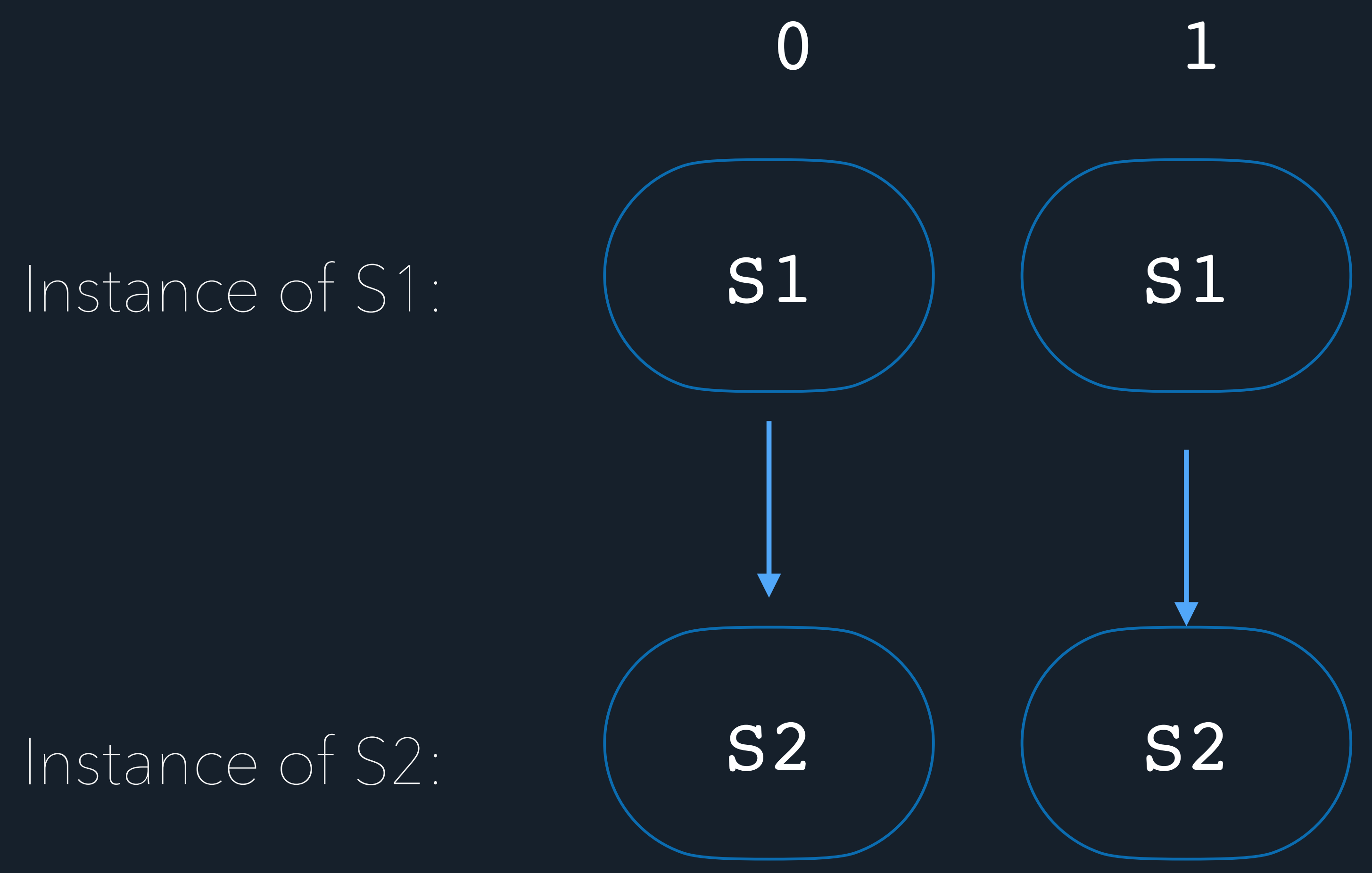
Flow dependency (Read After Write RAW)

- An instruction dependents on the result of a previous instructions

```
      for (int i=0; i<n; I++)
   S1   a[i] = b[i] + 1;
   S2   c[i] = a[i] + 1 ;
```

Unrolling the code

$$\begin{tikzcd}[column sep=small] & \texttt{I = 0} & & & & & \texttt{I = 1} \\ \texttt{S1} & \texttt{a[0]} \arrow[dr] & \texttt{= b[0] + 1;} & & \texttt{S1} & \texttt{a[1]} \arrow[dr] & \texttt{= b[1] + 1;} \\ \texttt{S2} & \texttt{c[0] =} & \texttt{a[0] + 1 ;} & & \texttt{S2} & \texttt{c[1] =} & \texttt{a[1] + 1 ;} \end{tikzcd}$$



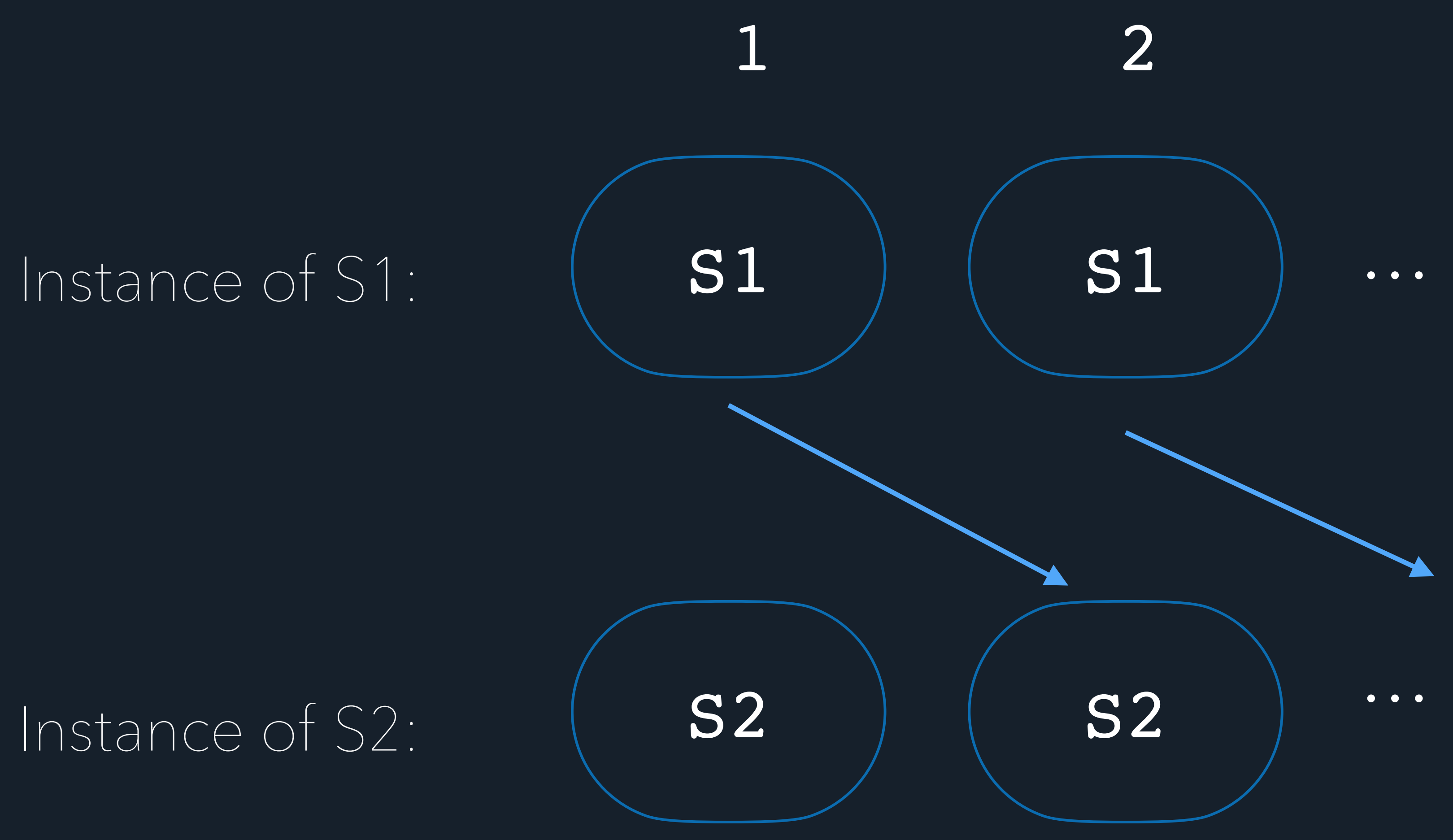
```
Loop independent dependence
```

Data dependencies

Anti dependency (Write After Read WAR)

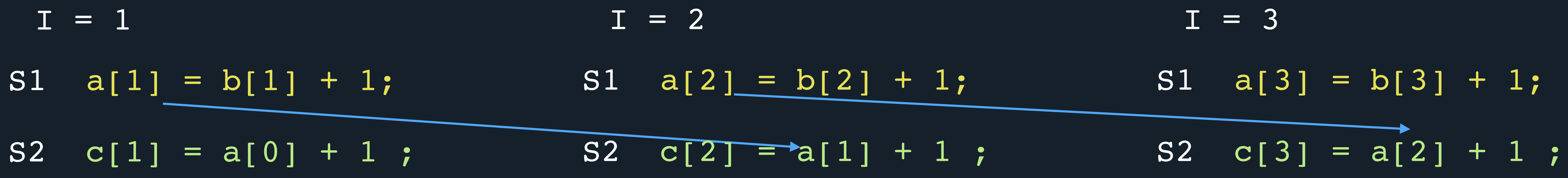
- An instruction dependents on the result of a previous instructions

```
     for (int I=1; i<n; I++)
S1   a[i] = b[i] + 1;
S2   c[i] = a[i-1] + 1 ;
```



Unrolling the code

Loop carried dependence



```
 I = 1                         I = 2                         I = 3

S1  a[1] = b[1] + 1;          S1  a[2] = b[2] + 1;          S1  a[3] = b[3] + 1;

S2  c[1] = a[0] + 1 ;         S2  c[2] = a[1] + 1 ;         S2  c[3] = a[2] + 1 ;
```

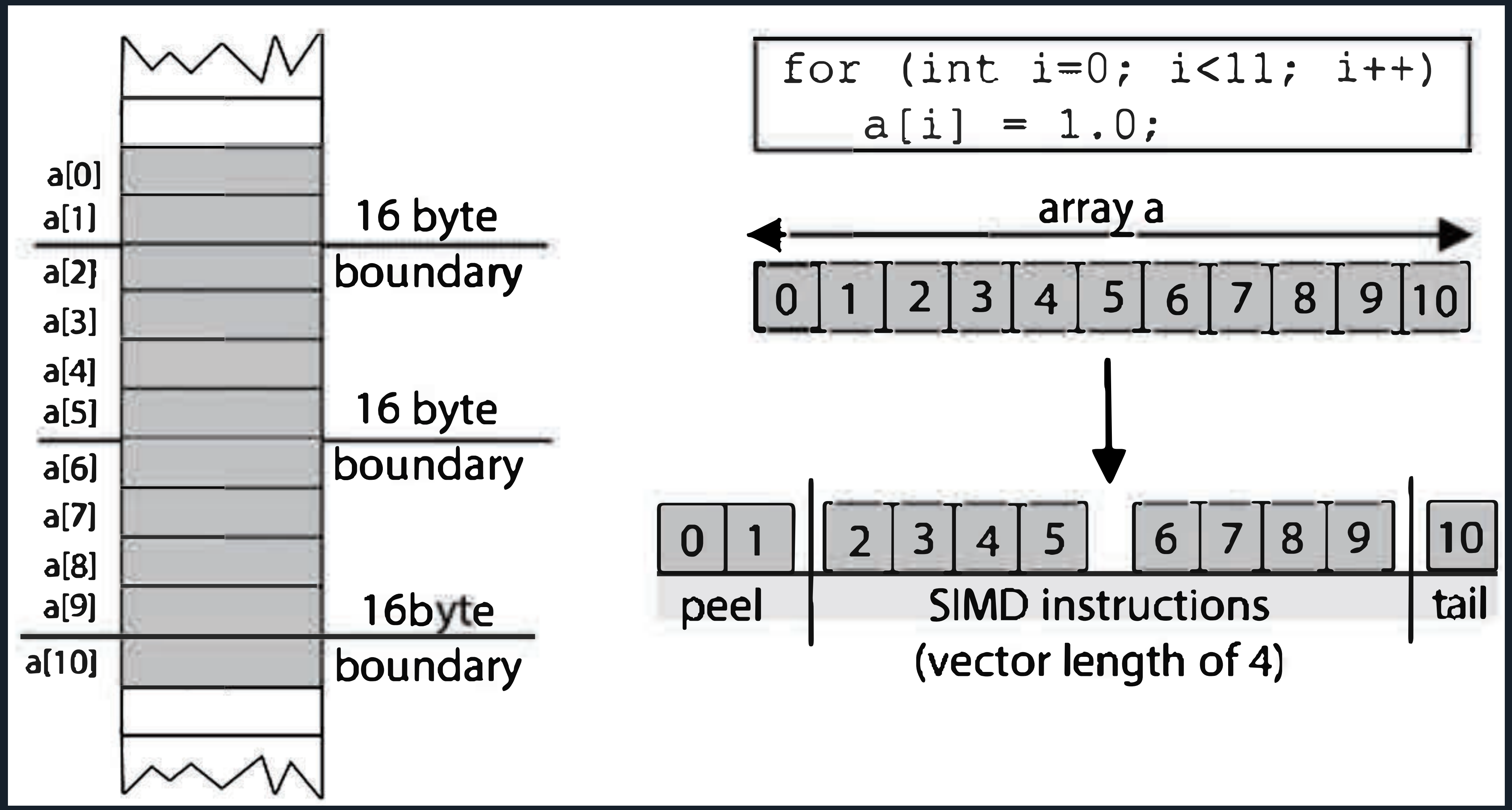
Data alignment

Vectorising data alignment is essential for performance

- To get the full benefit from SIMD, the starting address of the vectors may need to be aligned on the correct boundary.
- The address in memory must be a multiple of the vector length in bytes.

Unaligned data

- May require multiple data loading, multiple caches lines and multiple instructions
- Generates 3 different version of a loop: peel, kernel, tail


```
for (int i=0; i<11; i++)
   a[i] = 1.0;
```

OpenMP provides to pass on more information to the compiler to generate more efficient

Syntax of the SIMD construct in C/C++ and Fortran

OpenMP for SIMD vectorization

OpenMP SIMD Loop Construct

Vector parallelism is described with SIMD construct

- SIMD directives in OpenMP cut loop into chunks that fit a SIMD vector register
- No thread parallelization of the loop body

Syntax C/C++

```
#pragma omp simd [clause[[,] clause],…]
 for-loop
```

Syntax Fortran

```
!$omp simd [clause[[,] clause],…]
 do-loop
!$omp end simd
```

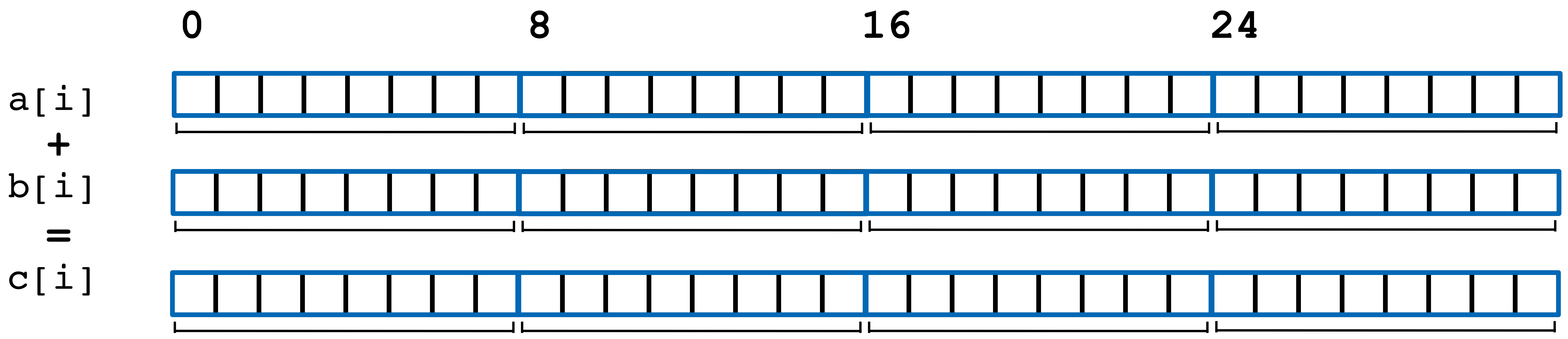
OpenMP SIMD: example

Syntax **C/C++**

```
#pragma omp simd
  for (int i=0; I<n; i++)
    c[i] = a[i] + b[I];
```

Syntax **Fortran**

```
!$omp simd
  Do i = 1, N
    c[i] = a[i] + b[i];
```



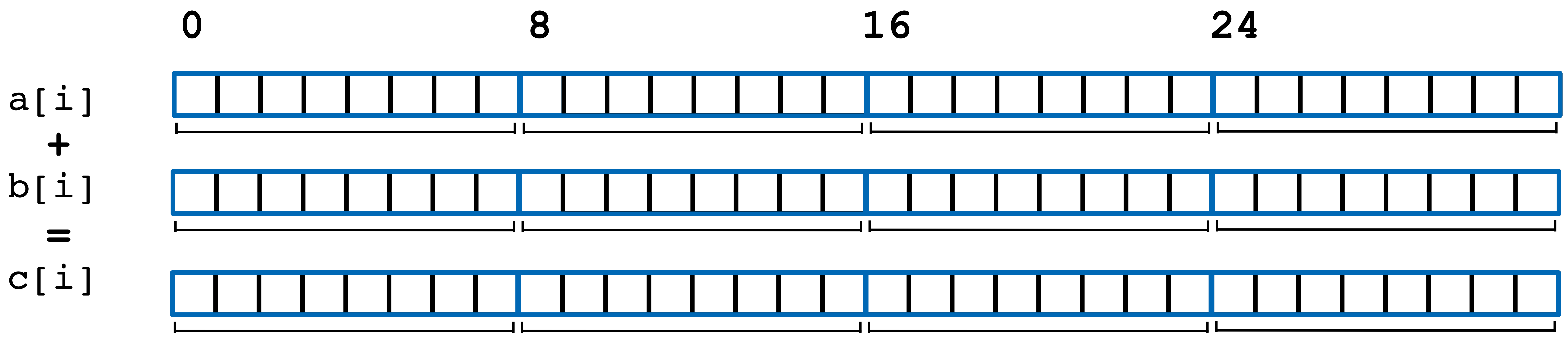
OpenMP SIMD worksharing construct

Syntax C/C++

```
#pragma omp for simd
  for (int i=0; I<n; i++)
    c[i] = a[i] + b[I];
```

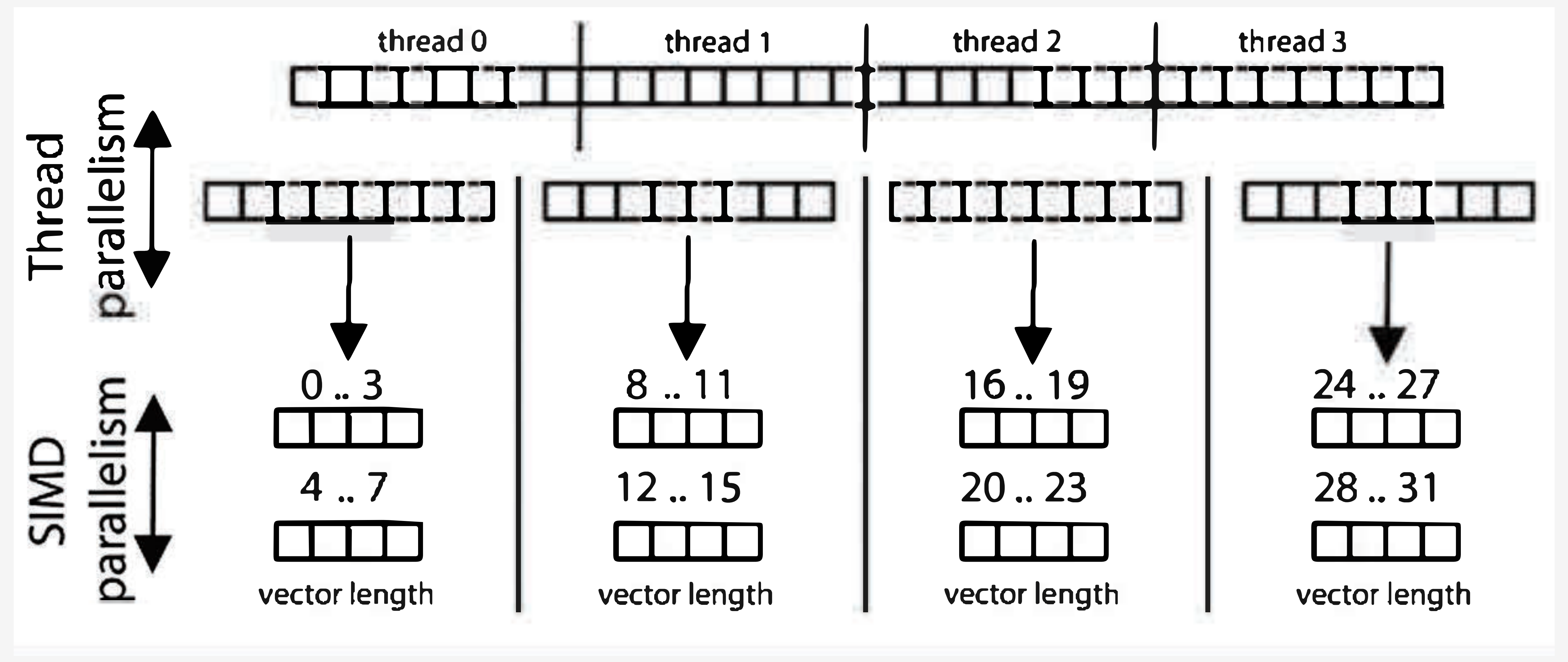
Syntax Fortran

```
!$omp parallel do simd
  Do i = 1, N
    c[i] = a[i] + b[i];
```



Which is performed first, vectorisation or thread-level parallelism

- **The distribution of loop iteration across the thread**
- **And then** Subdivide loop chunks to fit a SIMD vector register



Data Sharing Clauses

SIMD is tightly integrated with the OpenMP threading model

```
void simd_loop_private(double *a, double *b, double *c, int n)
{
  int i;
  double t1, t2;

  #pragma omp simd private(t1, t2)
  for (i=0; i<n; i++)
  {
    t1 = func1(b[i], c[i]);
    t2 = func2(b[i], c[i]);
    a[i] = t1 + t2;
  }
}
```

OpenMP SIMD construct instructs the compiler to generate a SIMD loop

- Loop iteration variable i is private
- One private instance will be created per SIMD lane
- The memory pointed to by a, b and c (the arrays) is shared
- The compiler is free to select the appropriate vector length that is suitable for the target architecture

Clauses supported by the SIMD construct

private *(list)*
lastprivate *(list)*
reduction *(reduction-identifier : list)*
collapse *(n)*
simdlen *(length)*
safelen *(length)*
linear *(list[:linear-step])*
aligned *(list[:alignment])*

Ensure SIMD execution legality

- All the usual data-sharing and reduction clauses can be applied.
- safelen(): distance between iterations where its safe to vectorise.

C/C++

```
#pragma omp simd safelen(4)
for ( int i=1; i<SIZE-4 ; i++ ) {
        A[i] = A[i] + A[i+4];
}
```

Fortran

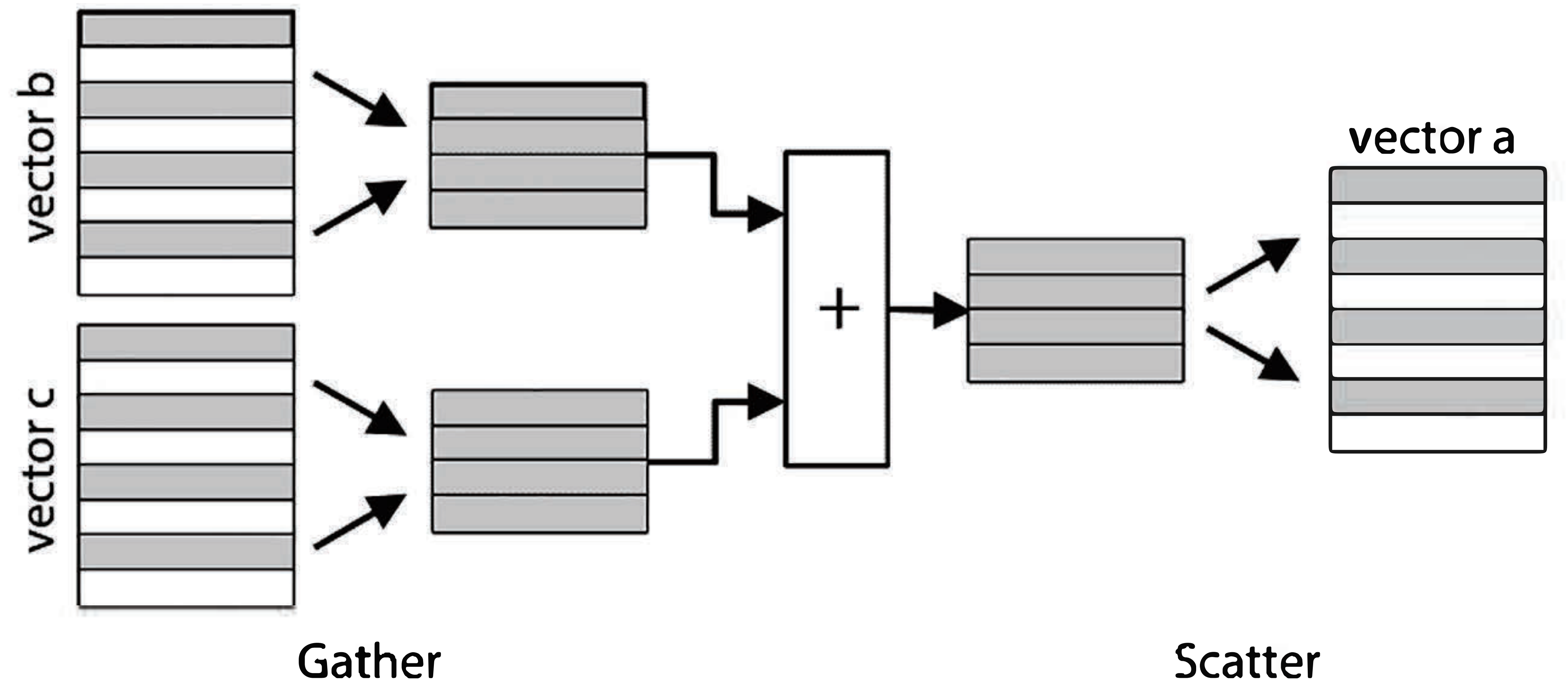
```
!$omp simd safelen(4)
do i=1,N-4
   A(i) = A(i) + A(i+4)
end do
!$omp end simd
```

- safelen(): preferred iterations to be performed concurrently as a vector.
- Specifying explicit vector lengths builds in obsolescence to the code as hardware vector lengths continually change — don't recommend using this clause.

SIMD linear Clause

- Arrays a and c are accessed through the loop variable i
- Array b is indexed through another variable j
- j has a linear relationship with the loop iteration variable i, which is incremented by 2 in each iteration, while j is incremented by 1

```
#pragma omp simd linear(j:1)
for (i=offset; i<n; i+=2)
  a[i] = b[j++] + c[i];
```



SIMD function vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

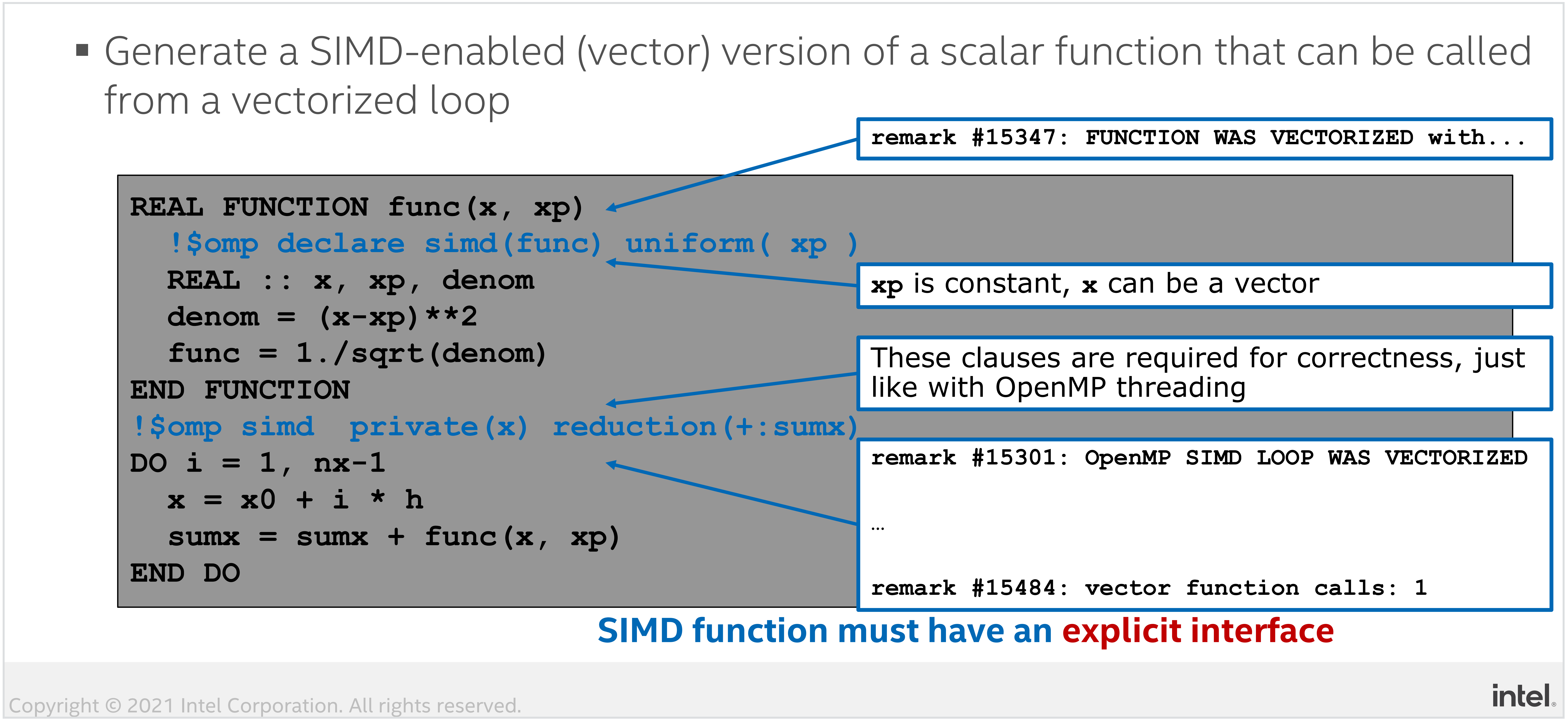
Syntax C/C++

```
#pragma omp declare simd [clause[[,] clause],...]
function-definition-or-declaration
```

Syntax Fortran

```
!$omp declare simd [clause[[,] clause],...]      ! Within function body
!$omp declare simd(proc-name-list)                ! At call site
```

Open DECLARE SIMD



- Generate a SIMD-enabled (vector) version of a scalar function that can be called from a vectorized loop

```
remark #15347: FUNCTION WAS VECTORIZED with...
```

```
REAL FUNCTION func(x, xp)
  !$omp declare simd(func) uniform( xp )
  REAL :: x, xp, denom
  denom = (x-xp)**2
  func = 1./sqrt(denom)
END FUNCTION
!$omp simd  private(x) reduction(+:sumx)
DO i = 1, nx-1
  x = x0 + i * h
  sumx = sumx + func(x, xp)
END DO
```

xp is constant, **x** can be a vector

These clauses are required for correctness, just like with OpenMP threading

```
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

…

remark #15484: vector function calls: 1
```

SIMD function must have an explicit interface

SIMD Function Vectorization

```
simdlen (length)
```
→ generate function to support a given vector length

```
uniform (argument-list)
```
→ argument has a constant value between the iterations of a given loop

```
inbranch
```
→ function always called from inside an if statement

```
notinbranch
```
→ function never called from inside an if statement

```
linear (argument-list[:linear-step])
aligned (argument-list[:alignment])
reduction (operator:list)
```

Same as before

Memory Access Collapsing

- OpenMP SIMD construct works by partitioning the logical iterations of the loop into SIMD chunks.
- This partitioning requires a single logical iteration space to be valid.

```
float A[4][4];
#pragma omp simd collapse(2)
for (int i = 0; i < 4; i++){
    for(int j = 0; j < 4; j++){
        A[i][j] = A[i][j] + 1;
    }
}
```

Correctly applied to the innermost loop in a set of nested loops

```
float A[4][4];
#pragma omp simd
for (int k = 0; k < 16; k++){

    int i = k / 4;
    int j = k % 4;
    A[i][j] = A[i][j] + 1;
}
```

The memory access to *A* is not linear with respect to the collapsed loop

Closing thoughts

- OpenMP SIMD construct instructs the compiler to generate a SIMD loop, which are present in several contemporary microprocessors
- The performance improvements from using SIMD instructions can be substantial
- Sometimes need to force the compiler to auto-vectorise (the correct) loop with the SIMD construct
- Check the compiler report before and after the check it did the right thing!
- Use declare simd and appropriate clauses if you need to create vectorised versions of functions

Some that are not explored here in detail?

- Conditional Calls to SIMD Functions
- SIMD Function Parameter Attributes
- SIMD with the Ordered Construct

List is quite long …

Exercise-1

a) Matrix multiplication
b) Add OpenMP SIMD constructs
c) Run and time it

Exercise-2

a) Estimate value of pi
b) Add OpenMP SIMD constructs
c) Run and time it

Grazie Mille!!

Feel free to reach me out
n.shukla@cineca.it