# OpenMP SIMD (Vectorization)

CSC Summer Institute

**October 9-11, 2023**

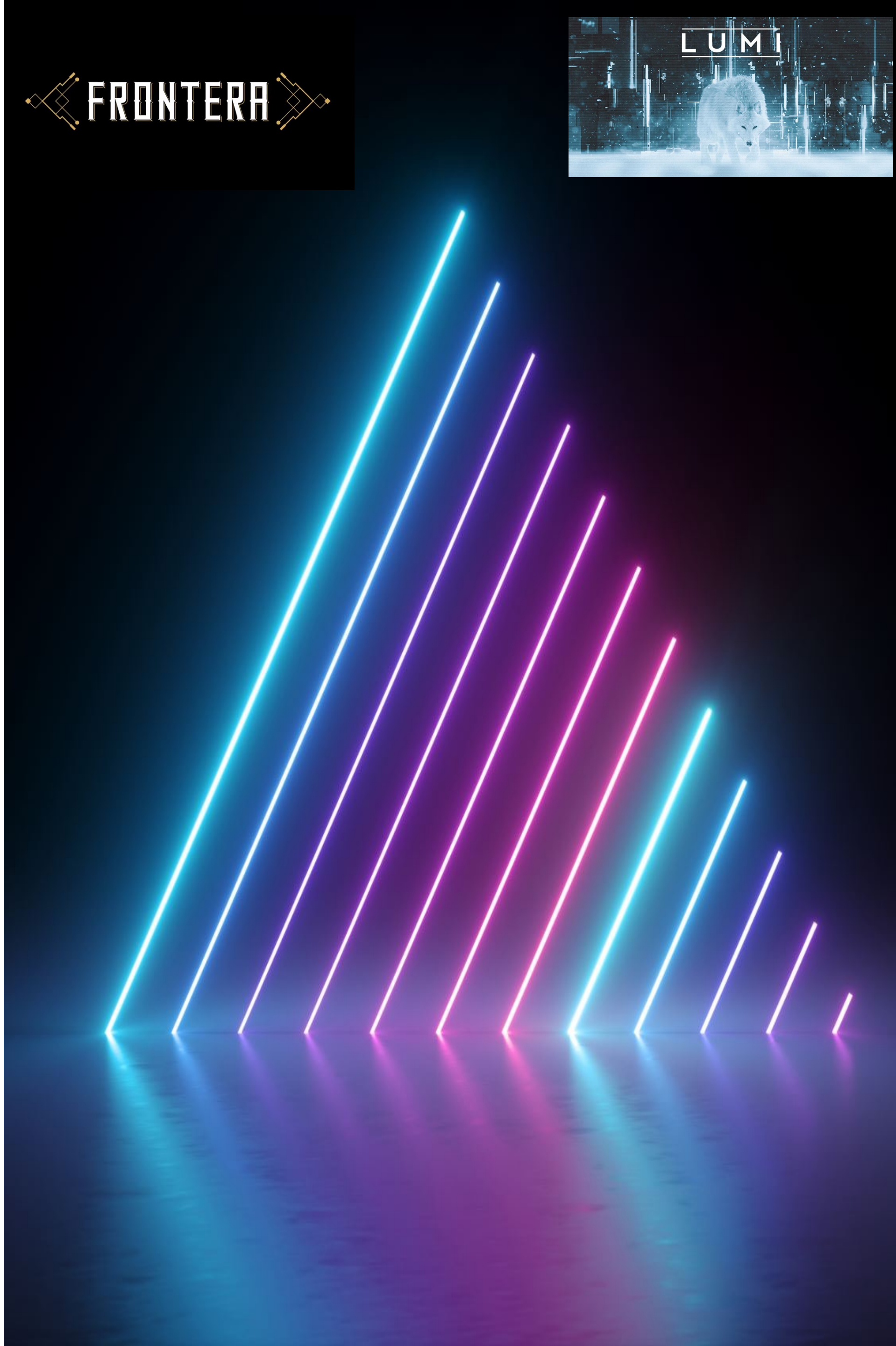
Kent Milfeld (TACC)

Emanuele Vitali (CSC)

TACC

CSC

ICT Solutions for Brilliant Minds



# OpenMP

- OpenMP is NOT just about threads
- OpenMP deals with multiple levels of Parallelism

target Directives

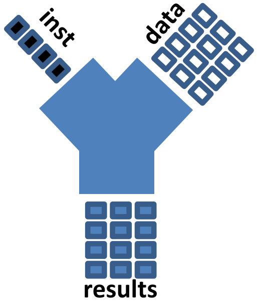
Across Devices

parallel do/for, task Directives

Across  Cores

simd Directives

Across SIMD Lanes



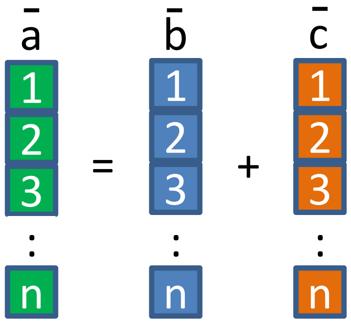
# Learning objective

- Vectorization and SIMD: what is this?
  - Programming Concept
  - Vector Hardware
- Code transformation
- SIMD Directives
  - SIMD loop directives
  - SIMD Enabled Functions
- SIMD + Threads
  - OpenMP
- Beyond Present Directives

# Programming Concept: SIMD (vectorization)

- Single Instruction, Multiple Data:
  ONE instruction applies same operation to multiple data concurrently

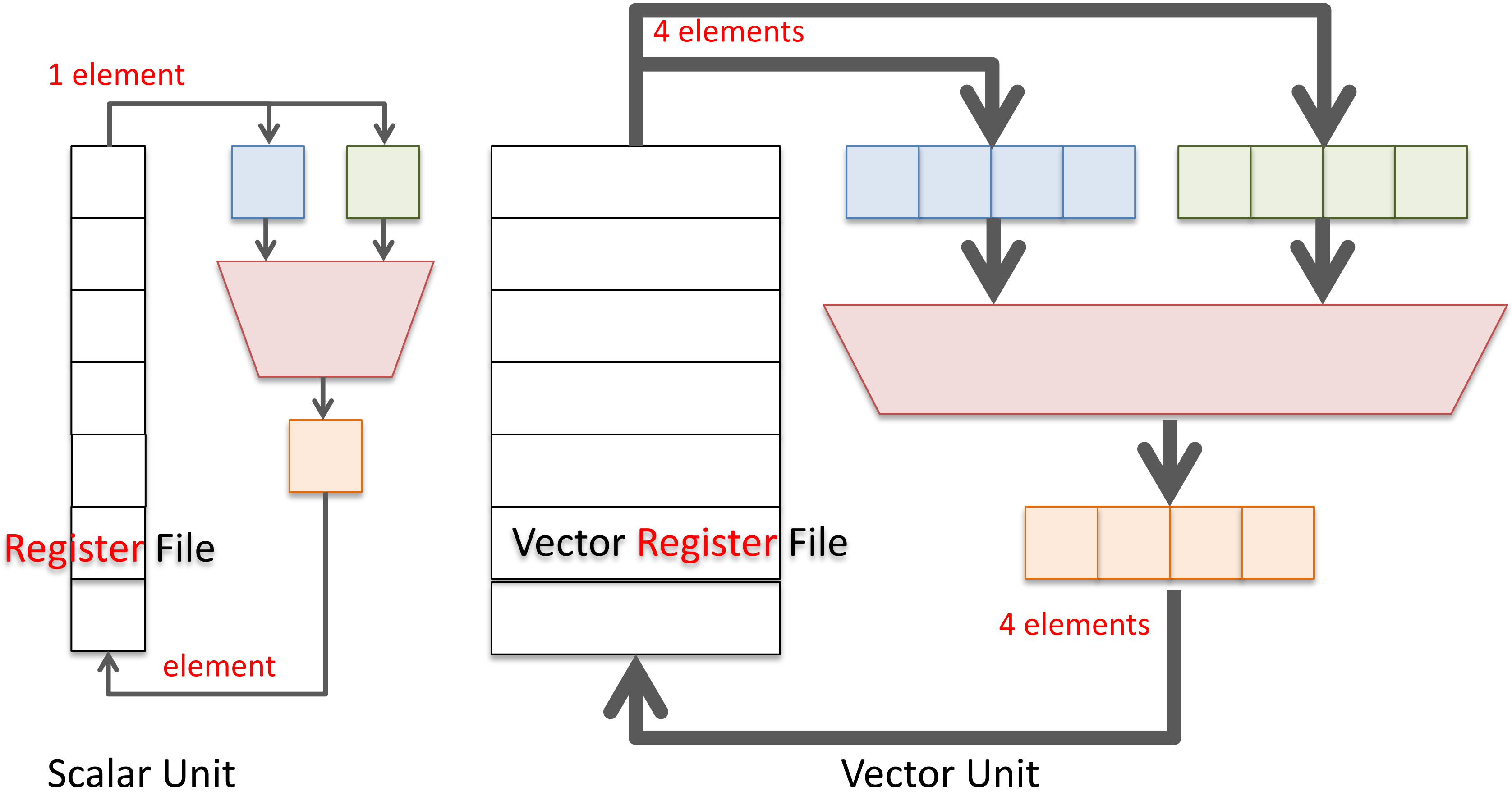
- Known as vectorization by scientific community.



```
do i = 1,n
   a(i) = b(i) + c(i)
end do
```

- SIMD directives = To specify what compilers may not determine.

# Vector Units (hardware)

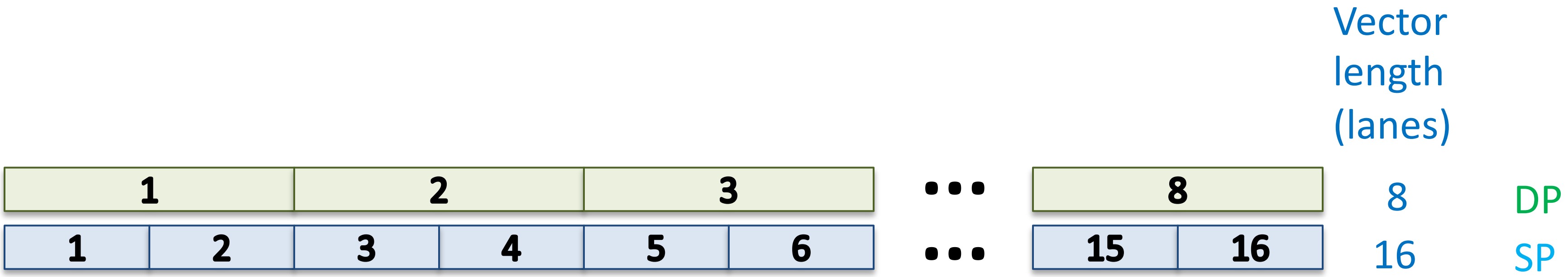


# Vector hardware

- Intel Skylake/KNL, Cascade Lake: 512 bits wide (Stampede2, Frontera)
  AVX512 instructions

DP= double precision computing = 64 bit floating point numbers

SP = single precision computing = 32 bits floating point number



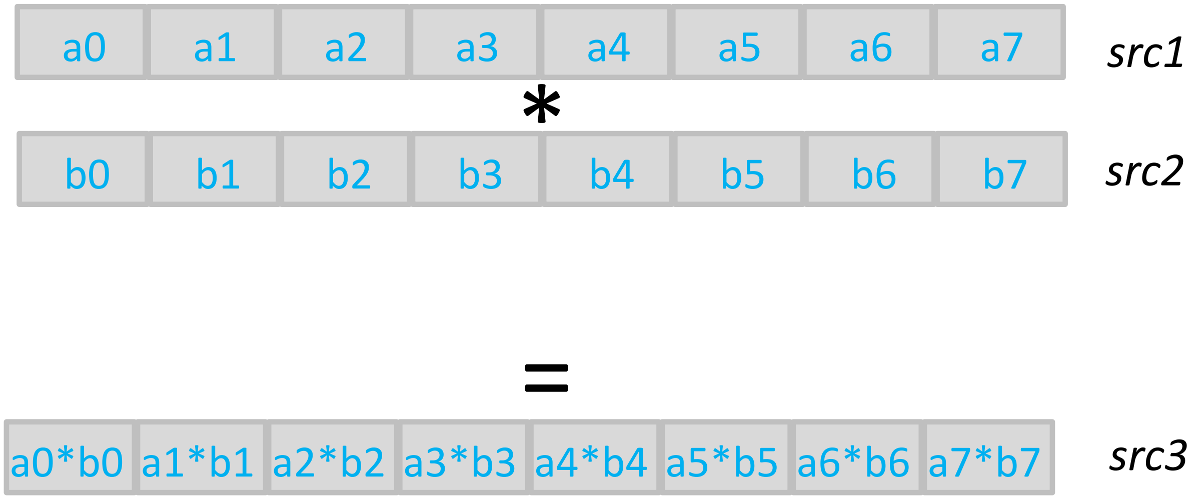
# Vector Instructions

AVX512 instructions set -- e.g. Intel (6-gen) and AMD (Zen-4 2x256):

MULT (MULTiply)

SIMD Instruction

vaddpd dest, *src1, src2*



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 | *src1* |
| | | | | * | | | | |
| b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 | *src2* |
| | | | | = | | | | |
| a0*b0 | a1*b1 | a2*b2 | a3*b3 | a4*b4 | a5*b5 | a6*b6 | a7*b7 | *src3* |

# Vector Instructions

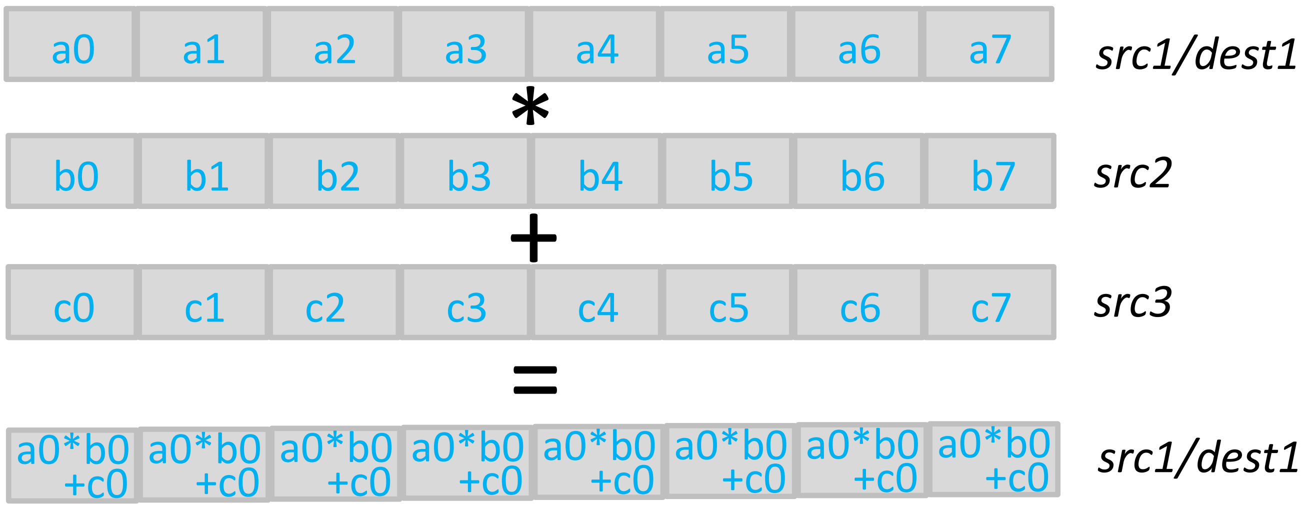
AVX512 instructions set -- e.g. Intel (6-gen) and AMD (Zen-4 2x256):

SIMD Instructions do more than just SIMD compute operations

SIMD Instruction

## FMA (Fused Multiply Add)

vfmadd213pd *src1/dest1*, *src2*, *src3*



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 | *src1/dest1* |
| | | | | * | | | | |
| b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 | *src2* |
| | | | | + | | | | |
| c0 | c1 | c2 | c3 | c4 | c5 | c6 | c7 | *src3* |
| | | | | = | | | | |
| a0*b0 +c0 | a0*b0 +c0 | a0*b0 +c0 | a0*b0 +c0 | a0*b0 +c0 | a0*b0 +c0 | a0*b0 +c0 | a0*b0 +c0 | *src1/dest1* |

# Vector Instructions

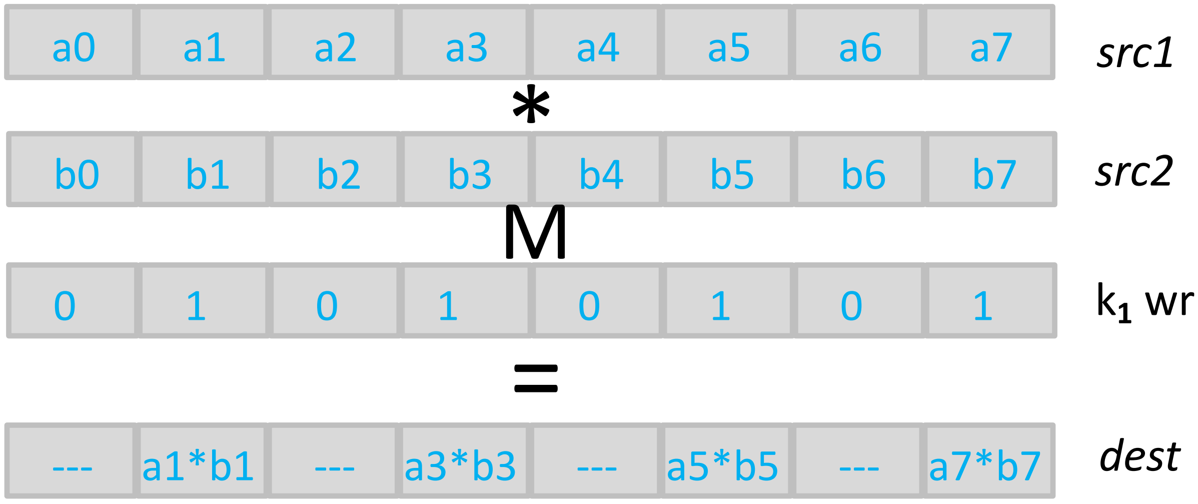
AVX512 instructions set -- e.g. Intel (6-gen) and AMD (Zen-4 2x256):

SIMD Instructions do more than just SIMD compute operations

## MASK (op)

SIMD Instruction

vaddpd *dest* {$k_1$}, *src2*, *src1*



|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 | *src1* |
|  |  |  |  | * |  |  |  |  |
| b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 | *src2* |
|  |  |  |  | M |  |  |  |  |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | $k_1$ write mask |
|  |  |  |  | = |  |  |  |  |
| --- | a1*b1 | --- | a3*b3 | --- | a5*b5 | --- | a7*b7 | *dest* |

# Vector hardware

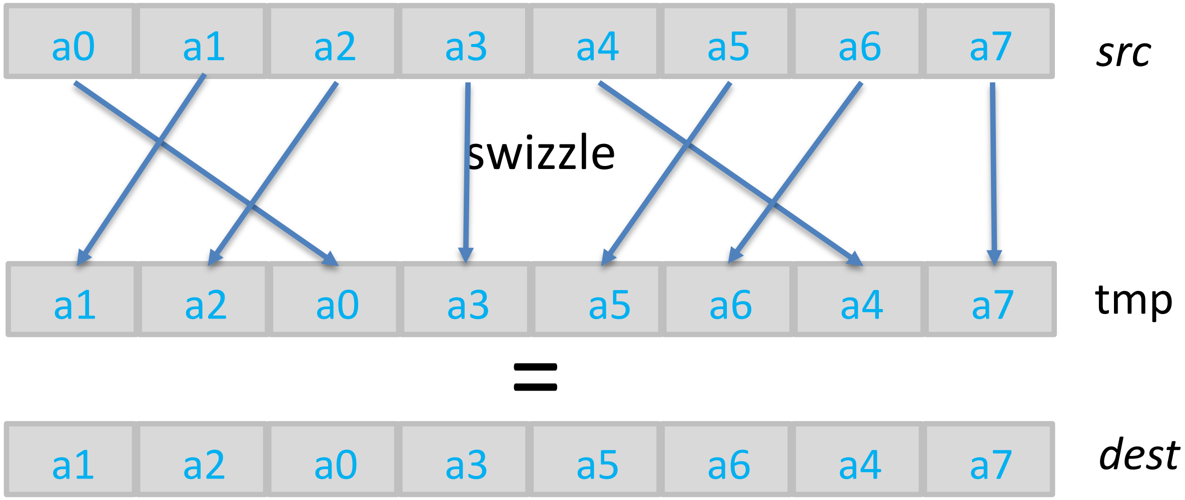
AVX512 instructions set -- e.g. Intel (6-gen) and AMD (Zen-4 2x256):

SIMD Instructions do more than just SIMD compute operations

SIMD Instruction

swizzle (and MOVe)

vmovapd *dest, src{dacb}*



a0 a1 a2 a3 a4 a5 a6 a7 *src*

swizzle

a1 a2 a0 a3 a5 a6 a4 a7 tmp

=

a1 a2 a0 a3 a5 a6 a4 a7 *dest*

TACC

CSC

ICT Solutions for Brilliant Minds

# How to vectorize the code?

- Use compiler options to set vector length (AVX<len>)
- The compiler will attempt to vectorize. (conservative approach: safety is utmost concern)
- Use compiler generated vectorization report to guide:
  - code changes
  - directives
- Use optimized libraries (BLAS, etc.)

# Other Optimization Targets

- Ideal Situation: Vector Units always in use
- vector (SIMD) length (register width)
- Other factors to consider besides vector (SIMD) length (register width).

| | |
|---|---|
| Striding: | 1 is best, random is worst |
| Masking: | Allows conditional execution, but you get lower performance |
| Caches: | Work with cached data (coherence, multiple levels) |
| Data arrangement: | AoS (Array of Structures) vs SoA (Structure of Arrays), Gather, Scatter, Permute Data |
| Alignment: | Avoid cache-to-register hiccups |
| Prefetching: | Sometimes users can improve the compiler’s result |

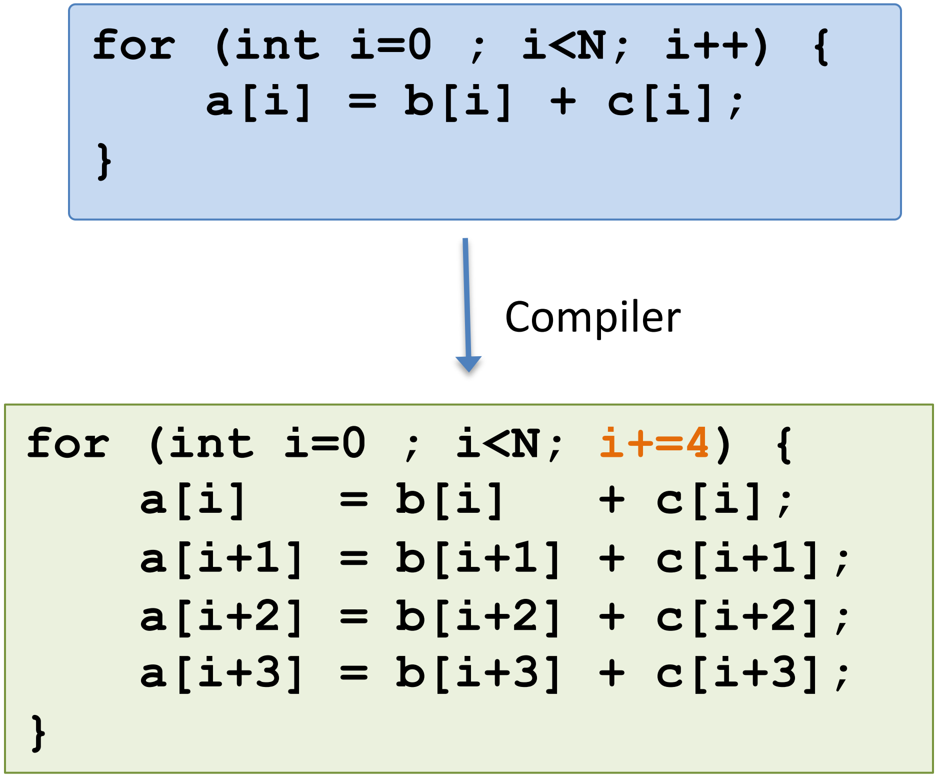
# SIMD Coding

- Exploit parallelism by applying the same operation to multiple data in parallel – with no dependences
- Typically applies to array operations in loops

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

```
do i=1, n
    a(i) = b(i) + c(i)
end do
```

# Transforming the code
# Example of 'Loop Unrolling'



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

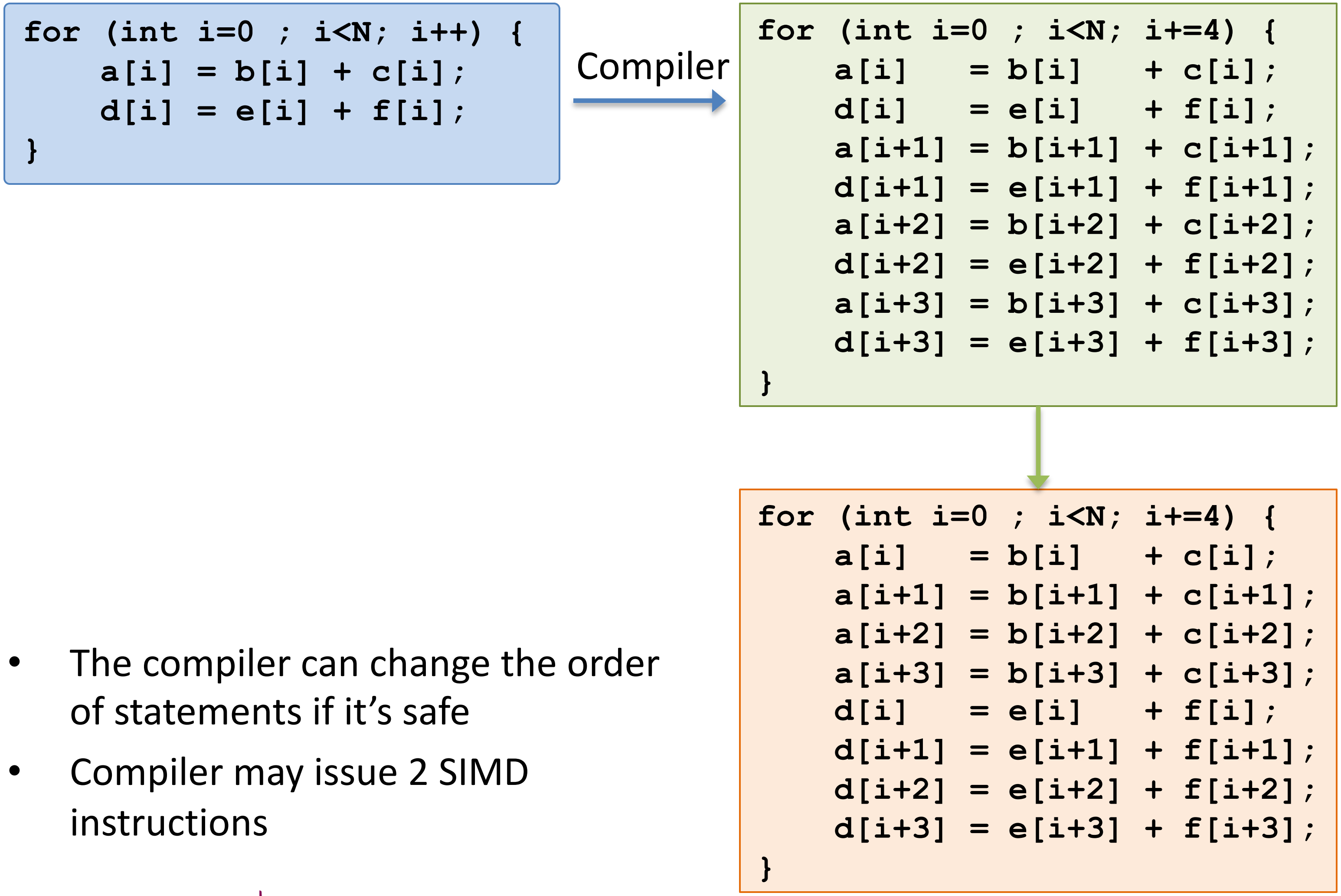
Compiler

```
for (int i=0 ; i<N; i+=4) {
      a[i]   = b[i]   + c[i];
      a[i+1] = b[i+1] + c[i+1];
      a[i+2] = b[i+2] + c[i+2];
      a[i+3] = b[i+3] + c[i+3];
}
```

Scalar execution:  4 instructions

Vector execution: 1 SIMD instruction (if the compiler can prove correctness)

# Transforming (more “complex”) code



```
for (int i=0 ; i<N; i++) {
    a[i] = b[i] + c[i];
    d[i] = e[i] + f[i];
}
```

```
for (int i=0 ; i<N; i+=4) {
    a[i]   = b[i]   + c[i];
    d[i]   = e[i]   + f[i];
    a[i+1] = b[i+1] + c[i+1];
    d[i+1] = e[i+1] + f[i+1];
    a[i+2] = b[i+2] + c[i+2];
    d[i+2] = e[i+2] + f[i+2];
    a[i+3] = b[i+3] + c[i+3];
    d[i+3] = e[i+3] + f[i+3];
}
```

```
for (int i=0 ; i<N; i+=4) {
    a[i]   = b[i]   + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
    d[i]   = e[i]   + f[i];
    d[i+1] = e[i+1] + f[i+1];
    d[i+2] = e[i+2] + f[i+2];
    d[i+3] = e[i+3] + f[i+3];
}
```

- The compiler can change the order of statements if it’s safe
- Compiler may issue 2 SIMD instructions

# Some loops cannot be vectorized

- Data dependencies in loop bodies
  - Dependencies introduced by algorithm
  - Dependencies introduced by programmer
- Complex loop bodies/complex code: vectorization may not be certifiable | “too costly”
- Coding complications
  - Loops with unknown number of iterations (while loop)
  - Loops with multiple exits
- General Function Calls inside loop adds complexity (inlining may help)

# What loops vectorize

- Of course, loops with independent iterations (vectorizable) ~~will~~ may vectorize!
  - It may happen by default (default optimization "O" level) --Intel
  - It may only occur above a certain optimization level (Cray PE 15.0:  available at -O1 for *ftn*, and -O2 for *CC* – and above)
  - Some vector analyzers are smarter that others, the level of optimization may affect vector capabilities.

- Your friends are the options that provide vectorization feedback
  - Cray PE loopmark

    Fortran: -hlist=m

    C/C++: `–fsave–loopmark`
  - Intel vector reports

    Fortran and C/C++: `–qopt–report–phase=vec`

# Helping the compiler (2) -- OpenMP

- OpenMP SIMD directive – It’s PORTABLE
- Directive is an instruction to the Compiler:
  - Assures independence of operations
  - “Do as I say, because I know what I’m doing”

# Helping the compiler -- OpenMP

- The OpenMP SIMD is applied to loops
  - Enables multiple iterations to be executed by SIMD instructions.
- The number of iterations that are executed concurrently is implementation defined
  - Each set of concurrent iterations is a SIMD chunk.
  - When **if** clause is false SIMD chunk size is 1.

C/C++

```
#pragma omp simd
for (…;…;…)
```

F90

```
!$omp simd
    do-loop
!$omp end simd  !optional
```

Can turn off vectorization with **if** clause.

# Helping the compiler -- OpenMP

- The SIMD can be a “stand-alone” construct (without being nested in a parallel do/for),

```
int main(){
#pragma omp parallel for simd
  for (…;…;…)
```

```
program main
!$omp parallel do simd
  do_loop
```

- or nested within a parallel do/for construct,

```
#pragma omp parallel for
#pramga omp simd
  for (…;…;…)
```

```
!$omp parallel do
!$omp simd
    do_loop
```

```
#pragma omp parallel for simd
  for (…;…;…)
```

```
!$omp parallel do simd
  do_loop
```

composite construct

# Helping the compiler -- OpenMP

- or nested within other loop constructs

  more about these later...

C/C++

```
#pragma omp taskloop simd
  for (…;…;…)
```

```
#pragma omp target distribute simd
  for (…;…;…)
```

F90

```
!$omp taskloop simd
  do_loop
```

```
!$omp target distribute simd
  do_loop
```

# OpenMP simd syntax

```
#pragma omp simd [clause][[,]clause]
     for (…;…;…)
```

C/C++

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

F90

clause:

| | |
|---|---|
| collapse(n) | nested loops (more work) |
| reduction(op: list) | vectorizes partials |
| safelen(length) | maximum distance between concurrent instructions |
| simdlen(length) | preferred number of iterations to be executed concurrently |
| linear(list[:linear-step]) | linear relationship with respect to iteration space |
| aligned(list[:alignment]) | |
| private(list) | |
| lastprivate(list) | |

# Why do we need SIMD directives

- Often independent-iteration loops don’t vectorize.
  - Reason for vectorization failure:  complicated indexing …
  - SIMD directive instructs the compiler to create SIMD operations for iterations of the loops.

vec-report=2 of intel compiler was helpful:

remark #15541: outer* loop was not auto-vectorized: consider using SIMD directive

```
void foo(double a[n][n], double b[n][n], int end){
#pragma omp simd                   //<-added after evaluating vec-report
for (int i=0 ; i<end ; i++) {
     a[i][0] = (b[i][0] - b[i+1][0]);
     a[i][1] = (b[i][1] - b[i+1][1]);
   }
}
```

*report refers to this single loop as the “outer” loop.

# OpenMP simd clauses

```
#pragma omp simd private(tmp) reduction(+:sum)
for (int i=0; i<n; i++) {
    tmp = b[i] + c[i] * alpha;
    sum += tmp;
}
```

C/C++

```
!$omp simd private(tmp) reduction(+:sum)
do i=1, n
    tmp = b(i) + c(i) * alpha
    sum = sum + tmp
end do
!$omp end simd
```

F90

# OpenMP simd clauses

```
off=PARAM  //PARAM always greater than 8

#pragma omp simd safelen(8)
for (int i=0; i<n; i++) {
    a[i] = a[i+off] + c[i] * alpha;
}
```

C/C++

```
off=PARAM !PARAM always gt 8

!$omp simd safelen(8)
do i=1, n
    a(i) = a(i+off) + c(i) * alpha
end do
!$omp end simd
```

F90

# SIMD enabled functions

- SIMDizable functions: can be invoked with either scalar or array elements
- Think of it as “inlining” with vector capability.

Consider:

```
double foo(double r, double s, double t);       // function definition
                                                // in another file
void driver (double R[N], double S[N], double T[N]){
   for (int i=0; i<N; i++){
     A[i] = foo(R[i],S[i],T[i]);
   }
}
```

# OpenMP declare simd syntax

- Applied to a function to create one or more versions of the function that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop

```
#pragma omp declare simd [clause][[,]clause]   (C/C++)
```

```
!$omp       declare simd [clause][[,]clause]   (F90)
```

clause:

| | |
|---|---|
| **simdlen(length)** | Preferred number of iterations to be executed concurrently |
| **linear(list[:linear-step])** | Objects in *list* have a linear relationship with respect to the iteration |
| **uniform(list)** | Objects in *list* have invariant value for all concurrent invocations |
| inbranch | Function will be always called from inside an ‘if’ block |
| notinbranch | Function will never be called from inside an ‘if’ block |
| aligned(list[:alignment]) | Objects in *list* are aligned to the number of bytes indicated |

# SIMD enabled functions

C/C++

```
double foo(double r, double s, double t); // function definition
                                          // in another file

void driver (double R[N], double S[N], double T[N]){
   for (int i=0; i<N; i++){
     A[i] = foo(R[i],S[i],T[i]);
   }
}
```

```
#pragma omp declare simd simdlen(4), notinbranch
double foo(double r, double s, double t);

void driver (double R[N], double S[N], double T[N]){
   #pragma omp simd
   for (int i=0; i<N; i++){
     A[i] = foo(R[i],S[i],T[i]);
   }
}
```

# SIMD enabled functions

C/C++

```
#pragma omp declare simd uniform(r,s,c) linear(i:1)
double foo(double* r, double* s, int i, double c) {
   return r[i] * s[i] + c;
}

#pragma omp declare simd uniform(c) linear(r,s:1)
double bar(double* r, double* s, double c) {
   return *r * *s + c;
}

void driver (double* r, double* s, double* res, double c){
   #pragma omp simd
   for (int i=0; i<N; i++){
     res[i] = foo(r, s, i, c);
   }
   #pragma omp simd
   for (int i=0; i<N; i++){
     res[i] = bar(&r[i], &s[i], c);
   }
}
```

# worksharing + SIMD syntax

- OMP Directives can Workshare and SIMDize a loop
  - Creates SIMD loop with chunk sizes in increments of the vector size.
  - Remaining iterations are distributed “consistently”.

combined directives

```
#pragma omp parallel for simd [clause][[,]clause]  (C/C++)
```

```
!$omp       parallel do  simd [clause][[,]clause]  (F90)
```

*clauses*: any do/for clause  any SIMD clause

# SIMD and threads – OpenMP worksharing

```
#pragma omp declare simd
double foo(double r, double s, double t);


#pragma omp parallel for simd
for (i=start; i<end; i++){
    foo(a[i], b[i], i);
}
```

# Summary

- OpenMP SIMD directive can be used to specify vectorization, and vectorization parameters.
- declare SIMD directive can be used to specify vectorization of functions (with similar SIMD clauses)
- SIMD directive can be used in conjunction with worksharing loops.