OpenMP* SIMD Programming

Dr.-Ing. Michael Klemm
Senior Application Engineer
Software and Services Group
(michael.klemm@intel.com)

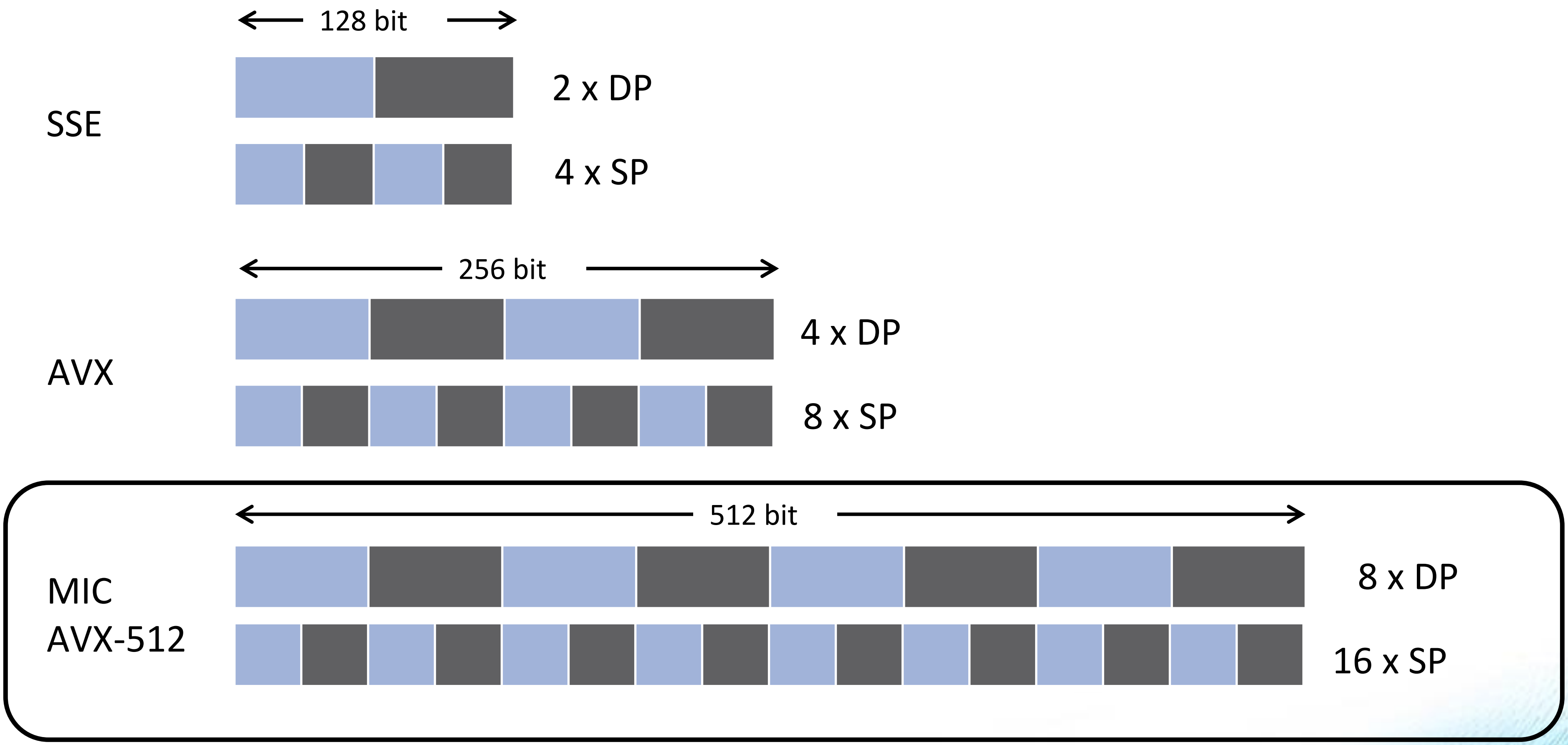
Legal Disclaimer & Optimization Notice

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Levels of Parallelism

Cluster	Group of computers communicating through fast interconnect
Coprocessors/Accelerators	Special compute devices attached to the local node through special interconnect
Node	Group of processors communicating through shared memory
Socket	Group of cores communicating through shared cache
Core	Group of functional units communicating through registers
Hyper-Threads	Group of thread contexts sharing functional units
Superscalar	Group of instructions sharing functional units
Pipeline	Sequence of instructions sharing functional units
Vector	Single instruction using multiple functional units

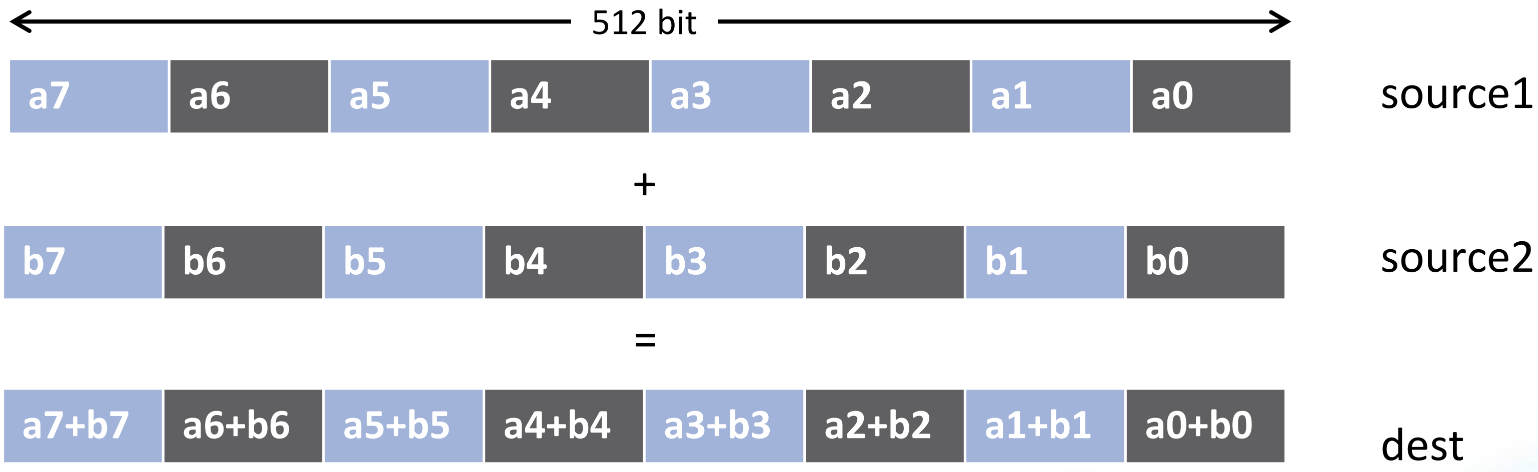
SIMD on Intel® Architecture



More Powerful SIMD Units

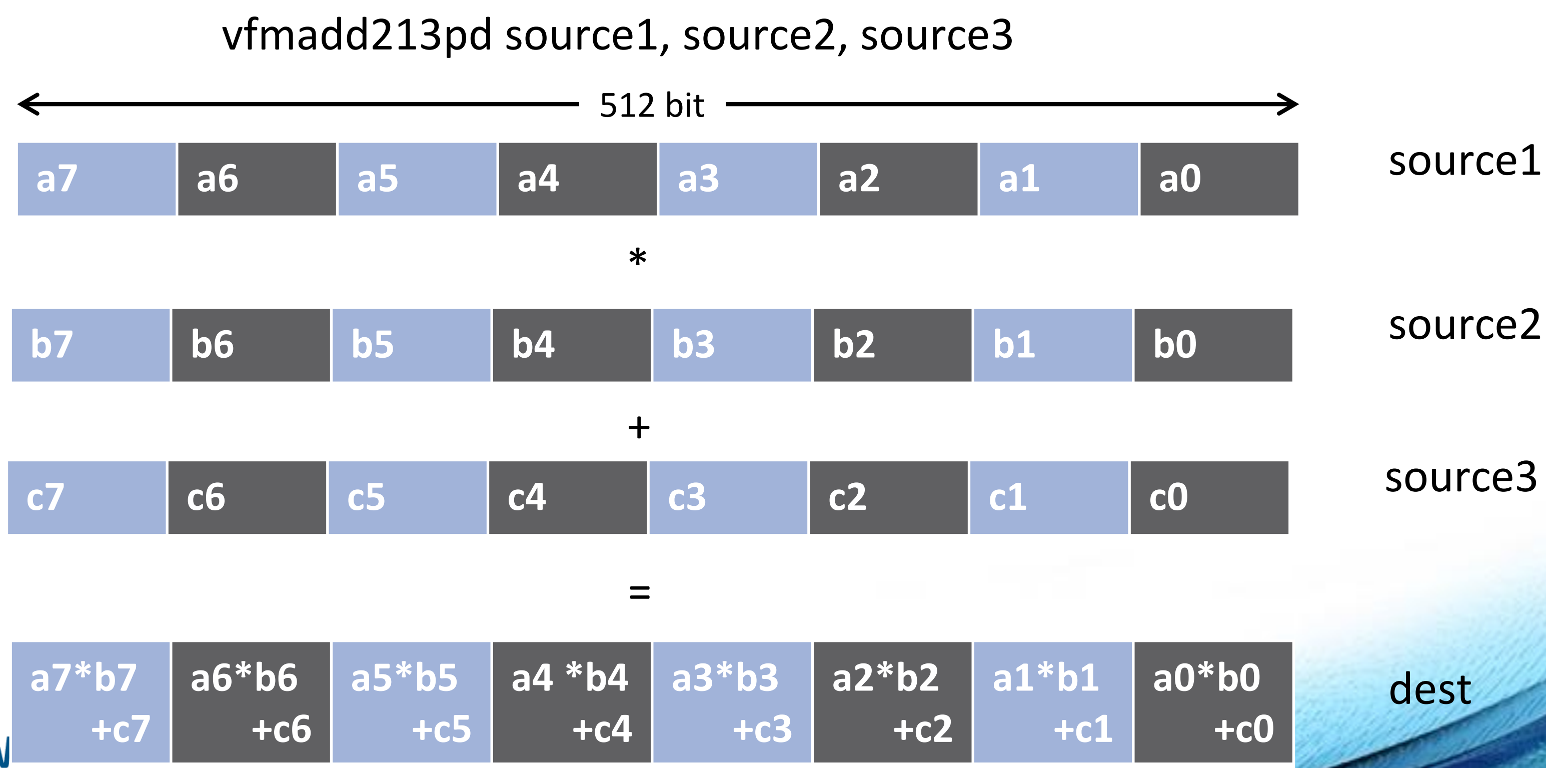
- SIMD instructions become more powerful on the Intel® Xeon Phi™ Processor

vaddpd dest, source1, source2



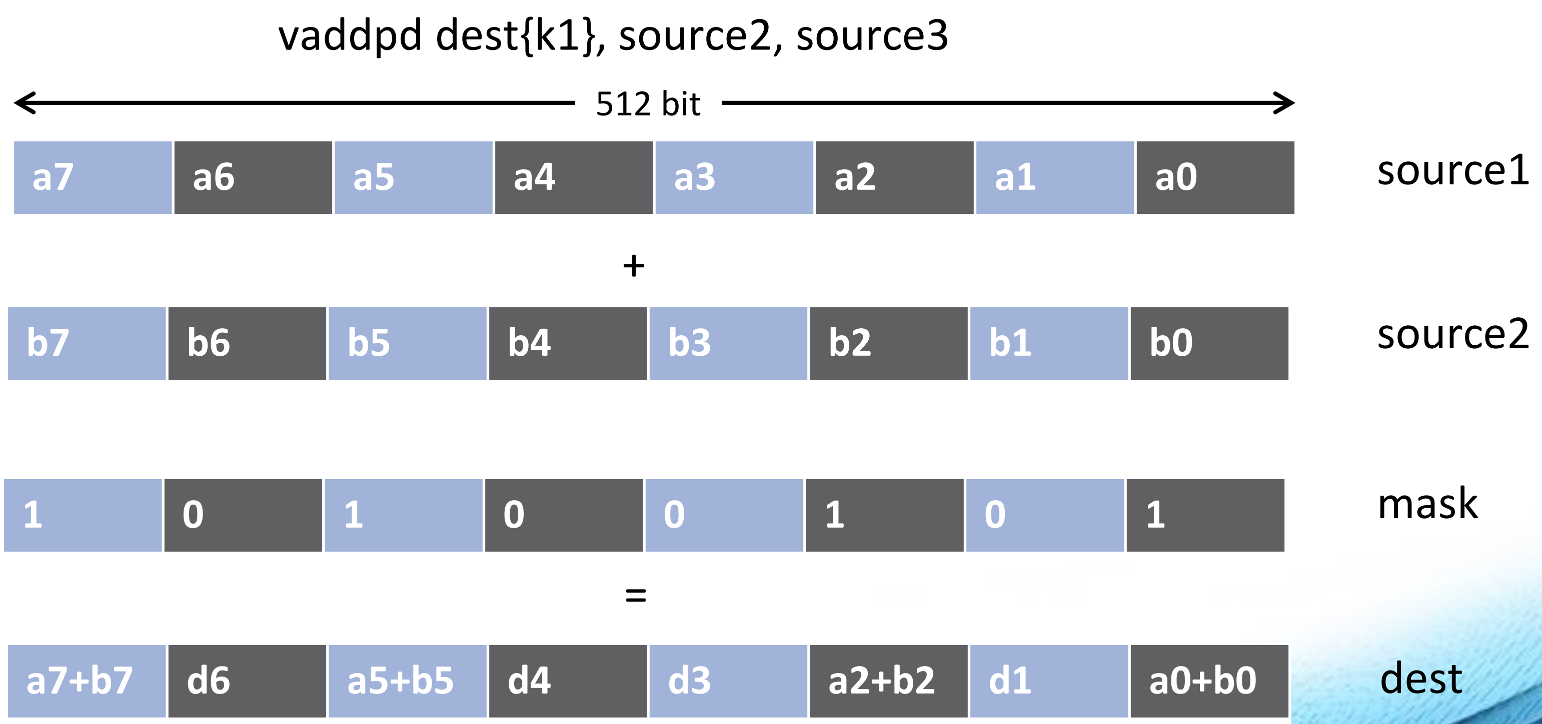
More Powerful SIMD Units

- SIMD instructions become more powerful on the Intel® Xeon Phi™ Processor



More Powerful SIMD Units

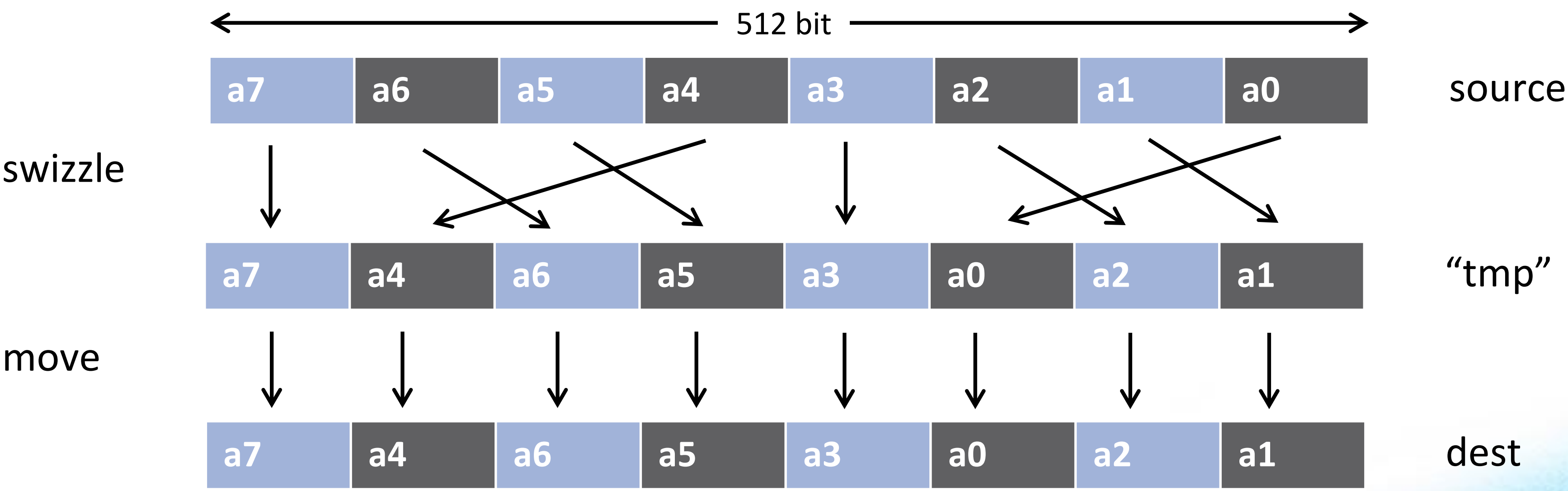
- SIMD instructions become more powerful on the Intel® Xeon Phi™ Processor



More Powerful SIMD Units

- SIMD instructions become more powerful on the Intel® Xeon Phi™ Processor

vmovapd dest, source{dacb}



- Auto vectorization only helps in some cases
  - 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
- Example: Intel® Composer XE
  - -vec (automatically enabled with **–O2**)
  - -qopt-report

Why Auto-vectorizers Fail

- Data dependencies
- Other potential reasons
  - Alignment
  - Function calls in 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 seems inefficient
- Many more … but less likely to occur

Example: Loop not Countable

- “Loop not Countable” plus “Assumed Dependencies”

```
typedef struct {
    float* data;
    size_t size;
} vec_t;

void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

In a Time before OpenMP 4.0

- Programmers had to rely on auto-vectorization…
- … or to use vendor-specific extensions
  - Programming models (e.g., Intel® Cilk™ Plus)
  - Compiler pragmas (e.g., `#pragma vector`)
  - Low-level constructs (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

You need to trust the compiler to do the "right" thing.

OpenMP SIMD Loop Construct

- Vectorize a loop nest
  - Cut loop into chunks that fit a SIMD vector register
  - No parallelization of the loop body

- Syntax (C/C++)

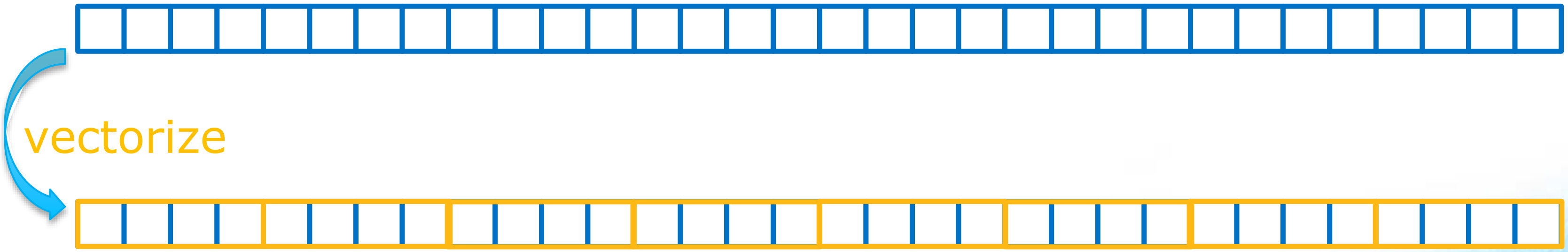
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

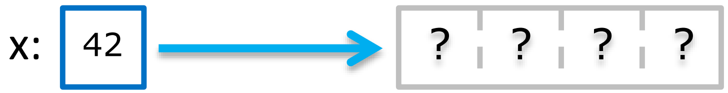
Example

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

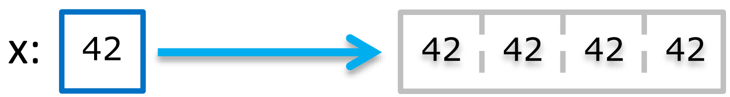


Data Sharing Clauses

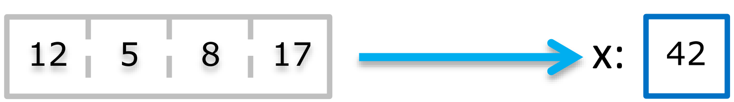
- `private(`*`var-list`*`):`
  Uninitialized vectors for variables in *var-list*



- `firstprivate(`*`var-list`*`):`
  Initialized vectors for variables in *var-list*



- `reduction(`*`op`*`:`*`var-list`*`):`
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct



SIMD Loop Clauses

- `safelen (`*`length`*`)`
  - Maximum number of iterations that can run concurrently without breaking a dependence
  - in practice, maximum vector length
- `linear (`*`list`*`[:`*`linear-step`*`])`
  - The variable's value is in relationship with the iteration number

    $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (`*`list`*`[:`*`alignment`*`])`
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- `collapse (`*`n`*`)`

Loop-Carried Dependencies

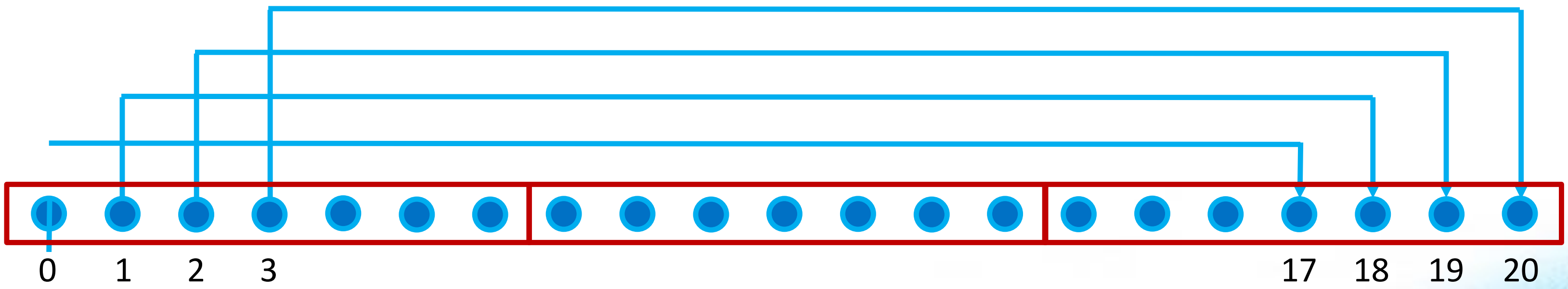
- Dependencies may occur across loop iterations
  - Loop-carried dependency
- The following code contains such a dependency:

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

- Some iterations of the loop have to complete before the next iteration can run
  - Simple trick: can you reverse the loop w/o getting wrong results?

Loop-Carried Dependencies

- Can we parallelize or vectorize the loop?
  - Parallelization: no
    (except for very specific loop schedules)
  - Vectorization: yes
    (if vector length is shorter than any distance of any dependency)



SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - Distribute a loop's iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register

- Syntax (C/C++)

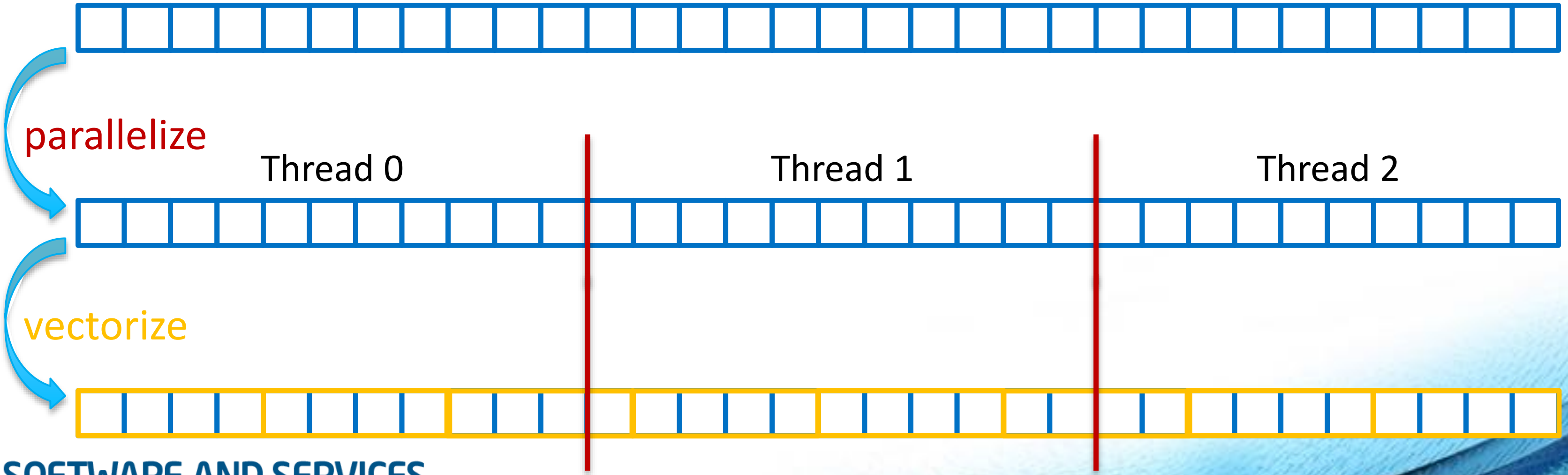
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

Example

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Be Careful What You Wish For…

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- You should choose chunk sizes that are multiples of the SIMD length
  - Remainder loops are not triggered
  - Likely better performance
- In the above example …
  - and AVX2, the code will only execute the remainder loop!
  - and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!

Schedule Modifiers

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(simd:static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- The new simd modifier automatically adjusts the chunk size to match it with the length of the SIMD register.
  - New chunk size becomes $\lceil chunksz/simdlen \rceil * simdlen$
  - AVX2: new chunk size will be 8
  - SSE: new chunk size will be 8

SIMD Function Vectorization

```
float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), 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],…]
[#pragma omp declare simd [clause[[,] clause],…]]
[…]
function-definition-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd (proc-name-list)
```

SIMD Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

```
vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}
```

```
vec8 distsq_v(vec8 x, vec8 y)
    return (x - y) * (x - y);
}
```

```
vd = min_v(distsq_v(va, vb, vc))
```

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

inbranch & notinbranch

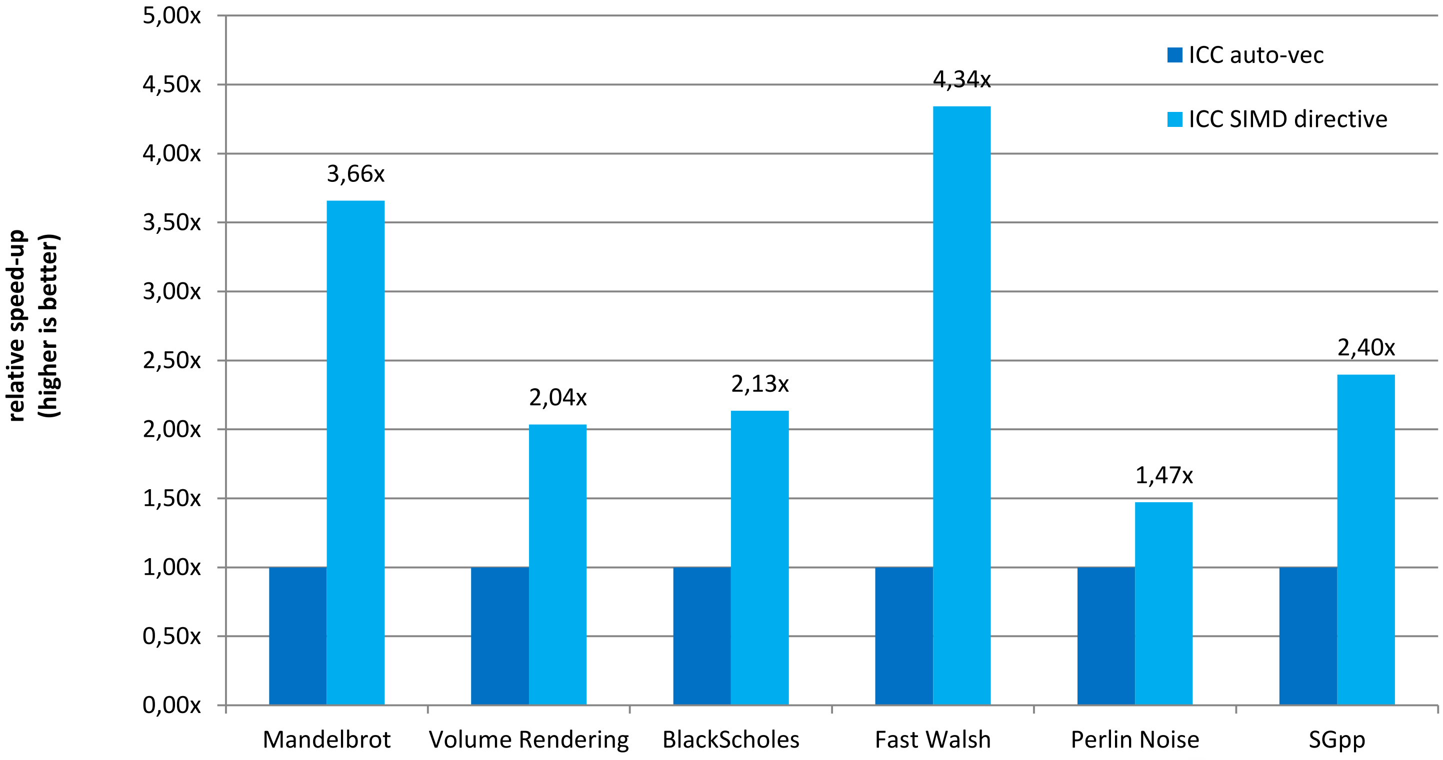
```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
#pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}
```

```
for (int i = 0; i < N; i+=8) {
    vcmp_lt &a[i], 0.0, mask
    b[i] = do_stuff_v(&a[i], mask);
}
```

SIMD Constructs & Performance



M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

intel®