



SIMD PROGRAMMING

Dr.-Ing. Michael Klemm

Senior Application Engineer
Developer Relations Division
michael.klemm@intel.com

Chief Executive Officer
OpenMP Architecture Review Board
michael.klemm@openmp.org

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2018, Intel Corporation. All rights reserved. Intel, the Intel logo, Atom, Xeon, Xeon Phi, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

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

OPENMP* ARCHITECTURE REVIEW BOARD

OpenMP Architecture Review Board

The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language **high-level parallelism** that is **performant, productive and portable**.



Interaction of OpenMP Bodies

Language Committee:

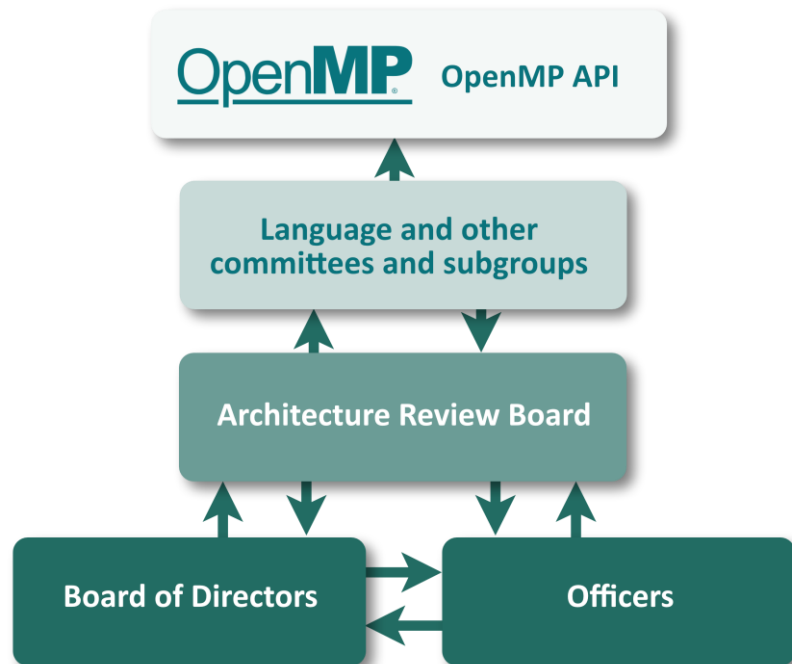
- Develops the OpenMP specification
- Technical Body of the ARB
- Several sub-committees focus on different topics (e.g., tasking)

Architecture Review Board

- Strategic alignment of the organization
- Owns the OpenMP specification

Board of Directors / Officers

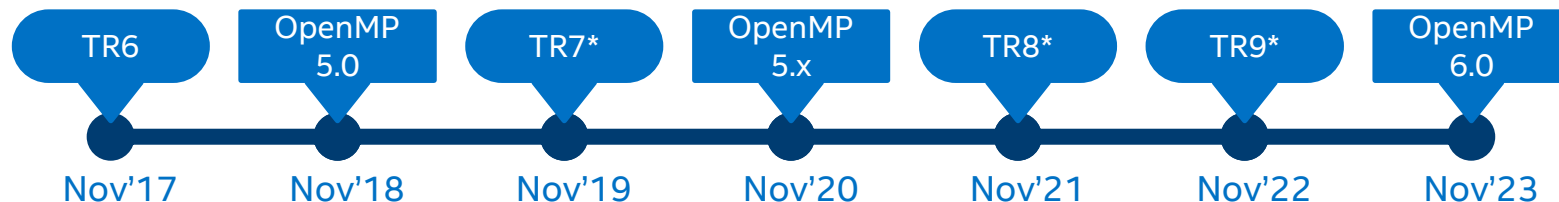
- BoD: Governance / regulatory body
- Officers: “run the business”



OpenMP Roadmap

OpenMP has a well-defined roadmap:

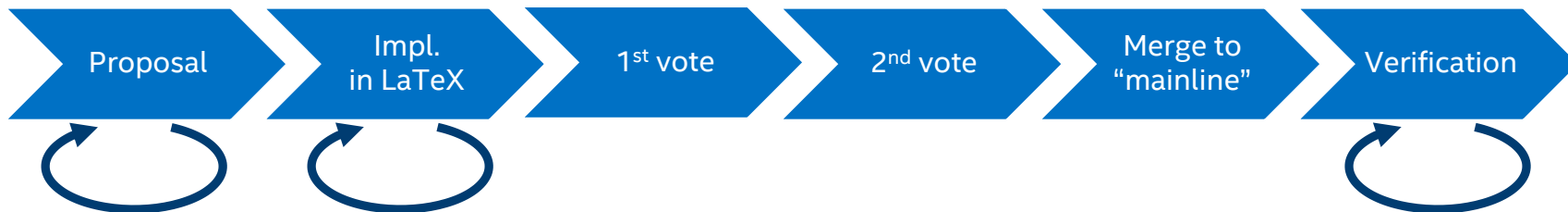
- 5-year cadence for major releases
- One minor release in between
- (At least) one Technical Report (TR) with feature previews in every year



* Numbers assigned to TRs may change if additional TRs are released.

Development Process of the Specification

Modifications of the OpenMP specification follows a (strict) process:

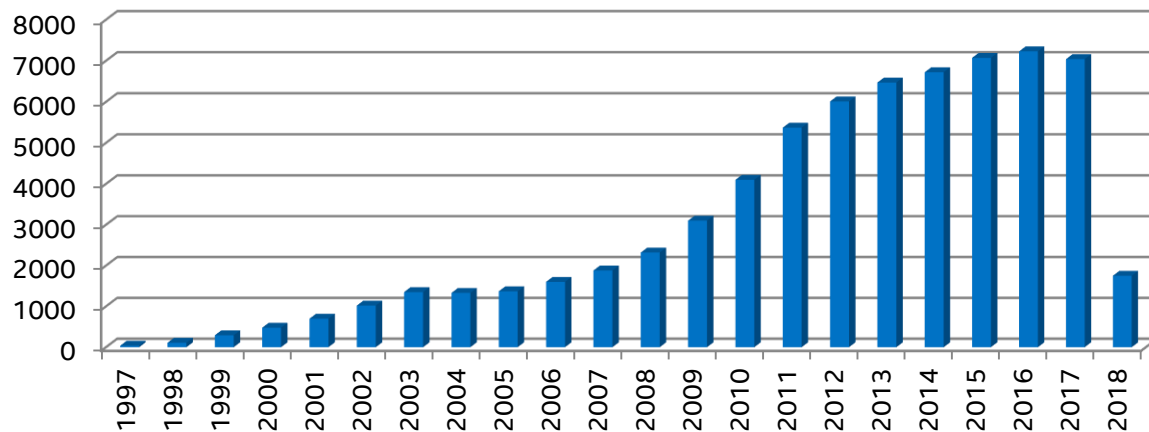


Release process for specifications:

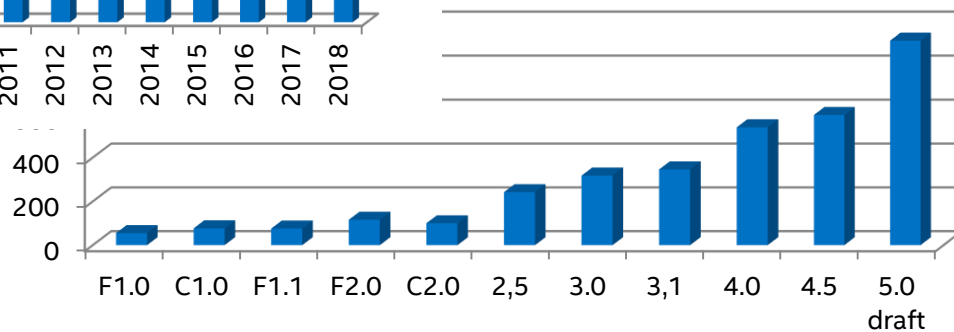


A Tiny Bit of Statistics

Google Scholar Hits



volume evolution



Find out more...



www.openmp.org

The OpenMP API specification for parallel programming



UK OpenMP Users' Conference 2018

21-22 May, 2018 - St Catherine's College, Oxford, UK.

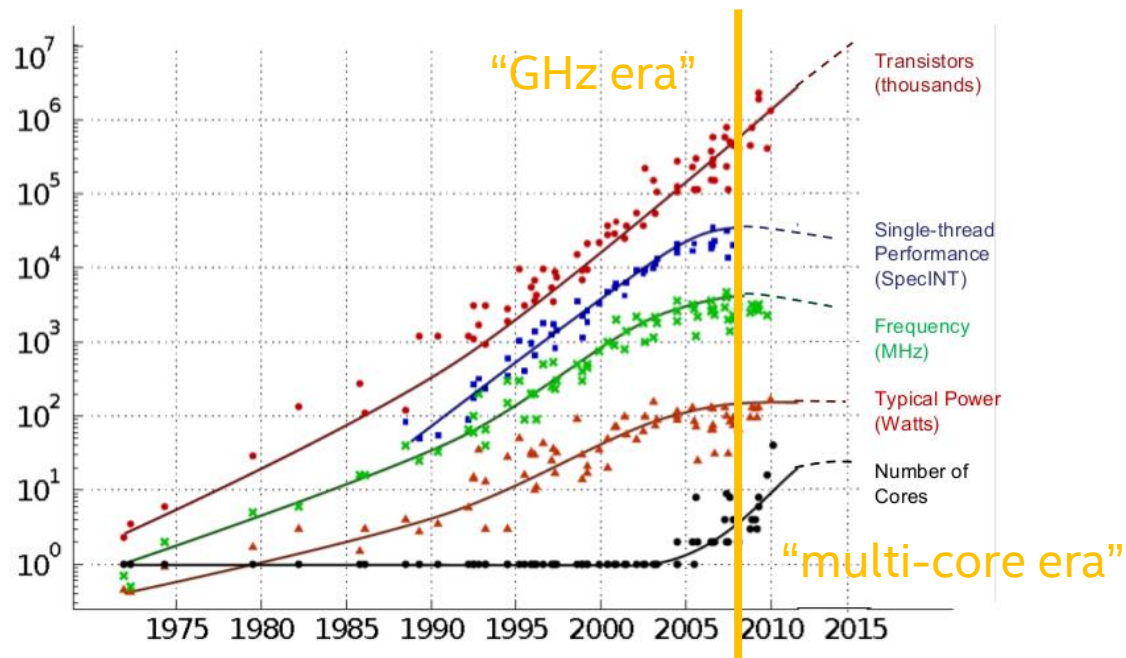
Tutorial & Conference Program Now Published

FIND OUT MORE

SIMD?

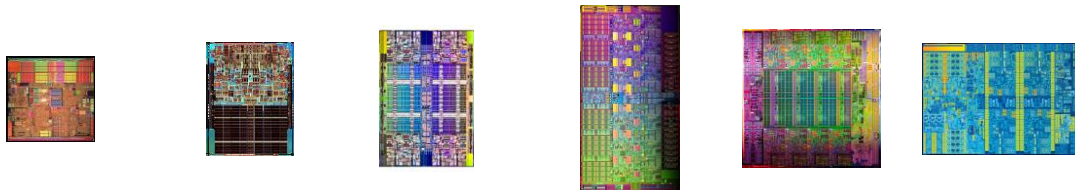
Moore's Law and Parallelism

35 YEARS OF MICROPROCESSOR TREND DATA



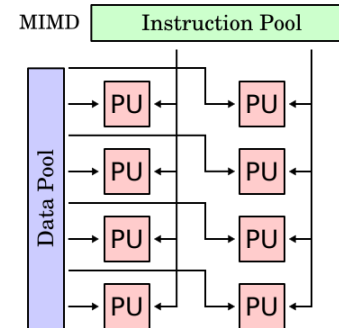
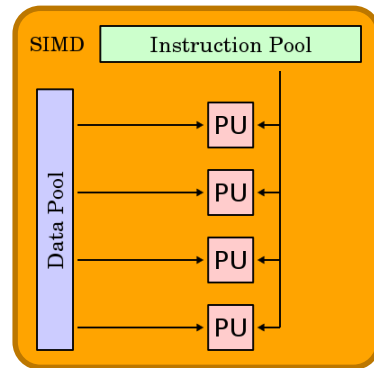
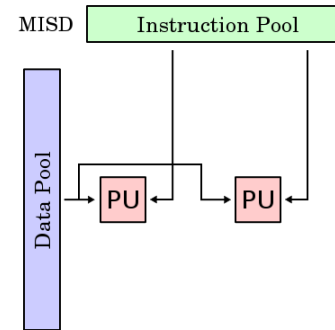
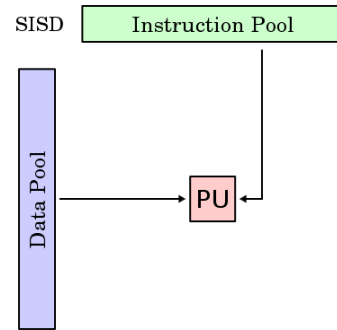
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Evolution of Intel Hardware



	64-bit Intel® Xeon® processor	Intel® Xeon® Processor 5100 series	Intel® Xeon® Processor 5500 series	Intel® Xeon® Processor 5600 series	Intel® Xeon® Processor E5-2699v3	Intel® Xeon® Processor 8180
Frequency	3.6 GHz	3.0 GHz	3.2 GHz	3.3 GHz	2.3 GHz	2.5 GHz
Core(s)	1	2	4	6	18	28
Thread(s)	2	2	8	12	36	56
SIMD width	128b (2 clock)	128b (1 clock)	128b (1 clock)	128b (1 clock)	256b (1 clock)	512b (1 clock)

SIMD – Flynn's Taxonomy Recap



SIMD Processors

ARM*

- Advanced SIMD, NEON*
- Scalable Vector Instructions

MIPS*

- MIPS SIMD Architecture

IBM POWER*

- Vector Multimedia Extension
- Vector Scalar Extension

OpenRISC*

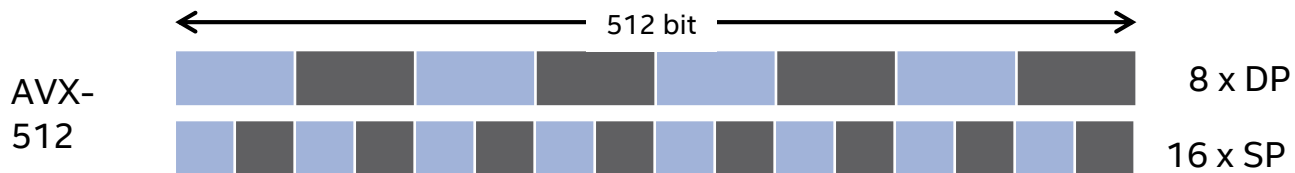
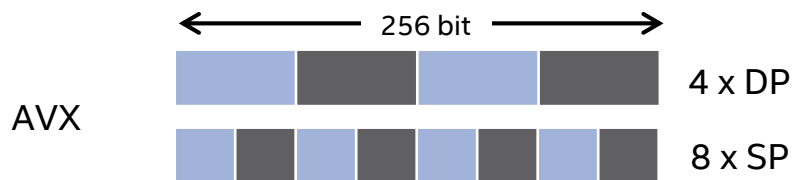
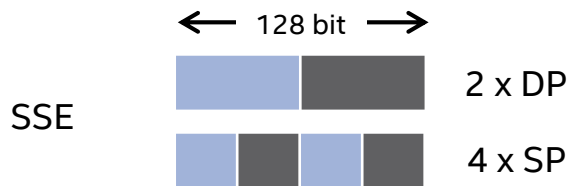
- ORV DX64

Oracle* SPARC*

- Visual Instruction Set

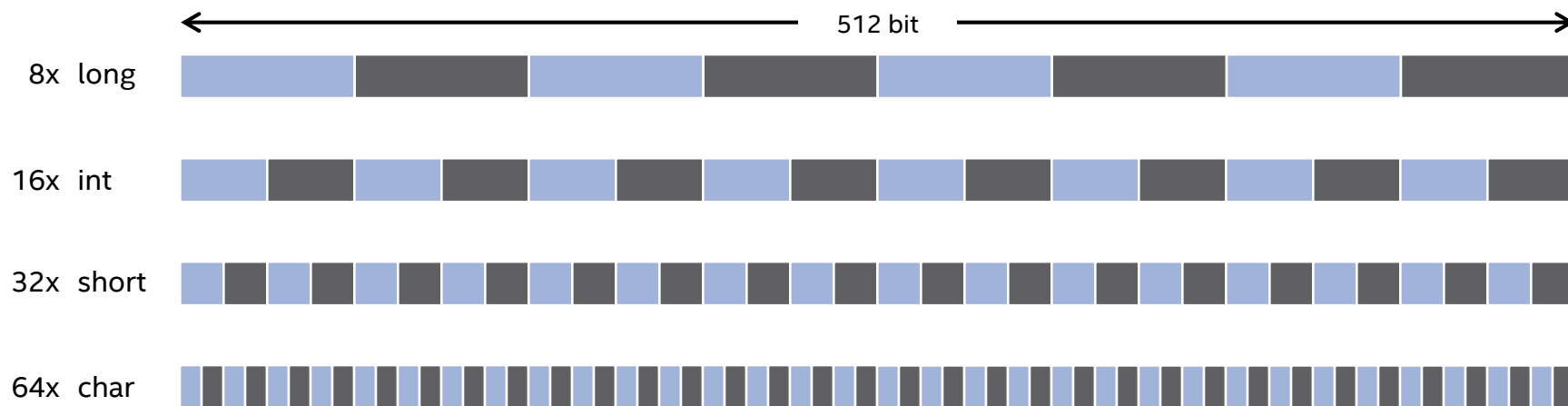
... and many more

Evolution of SIMD on Intel® Architectures



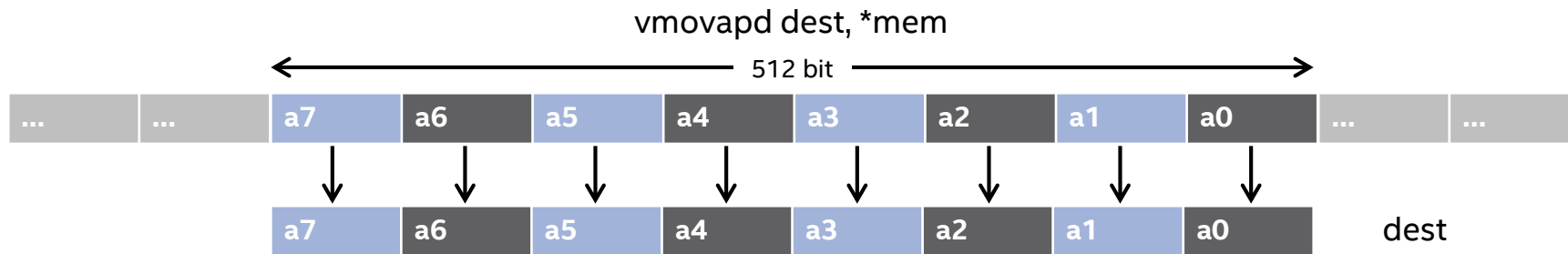
SIMD Instructions – Data Types

SIMD instruction sets typically support many more data types



SIMD Instructions – Load and Store

Load and store operations transfer data from memory/caches to the registers

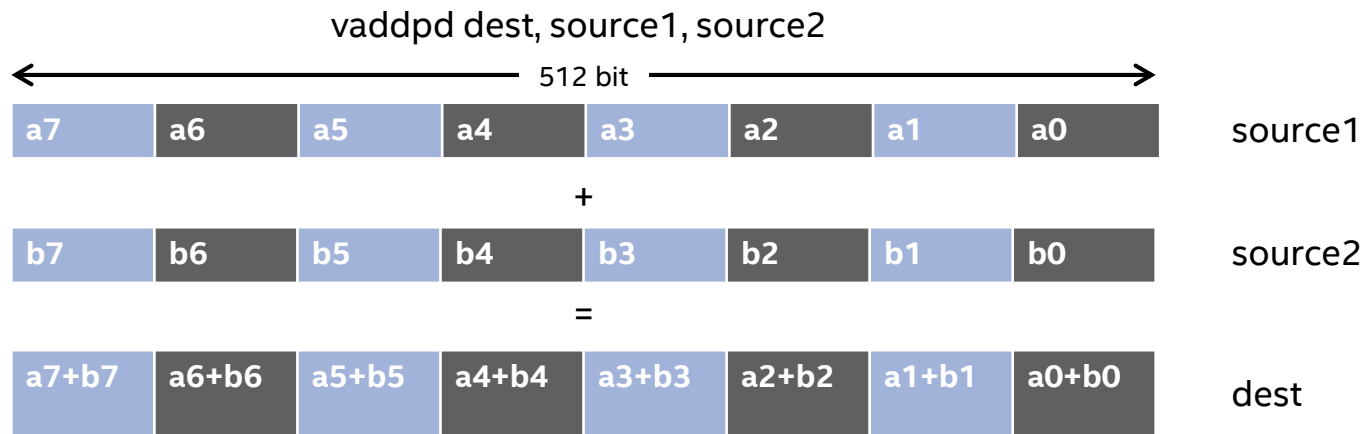


Loads/stores can happen:

- explicitly through load/store instructions, or
- implicitly through memory operands of other instructions.

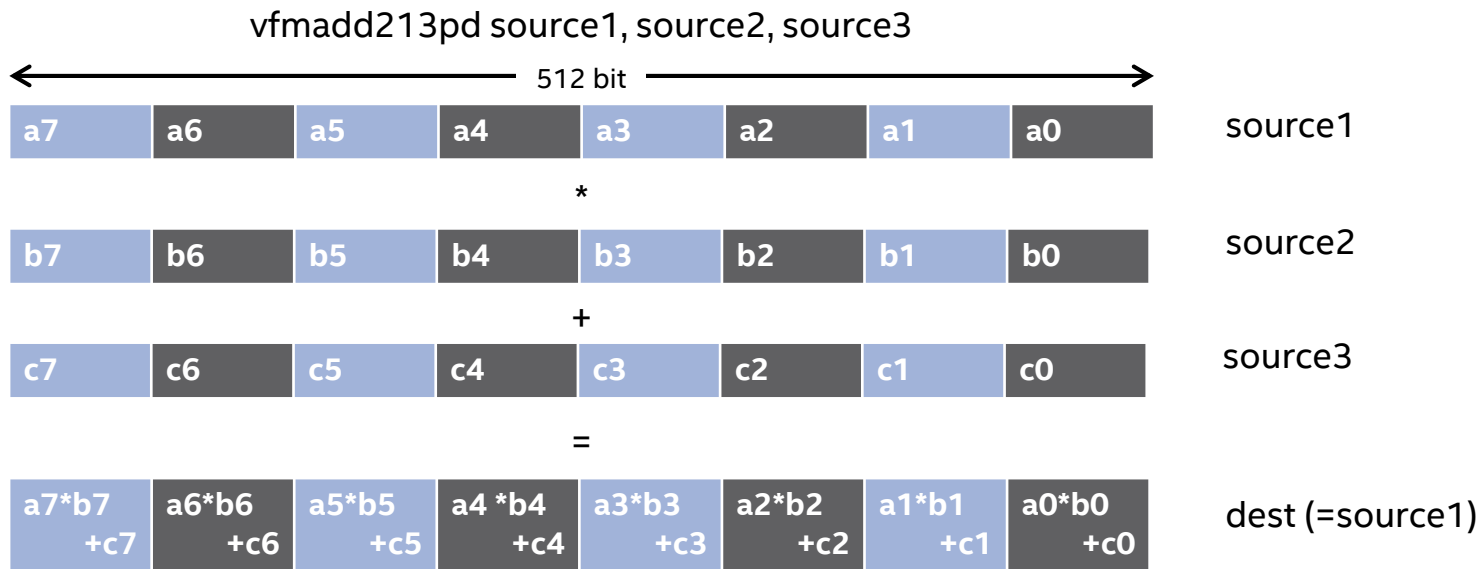
SIMD Instructions – Simple Arithmetic Instructions

Operations work on each individual SIMD element



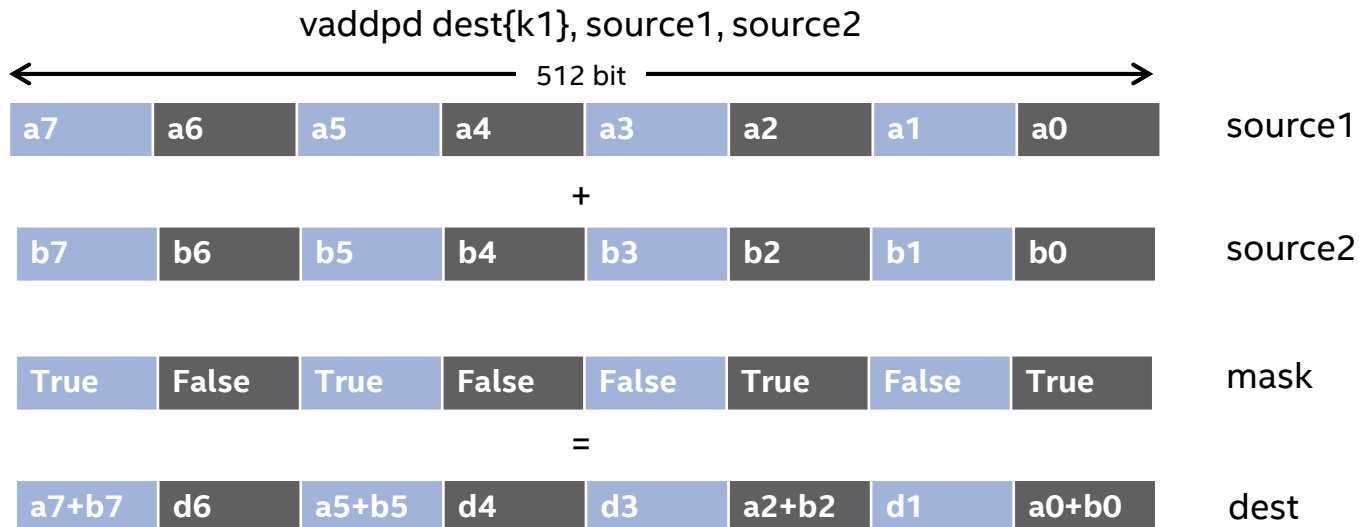
SIMD Instructions – Fused Instructions

Two operations (e.g., multiply & add) fused into one SIMD instruction



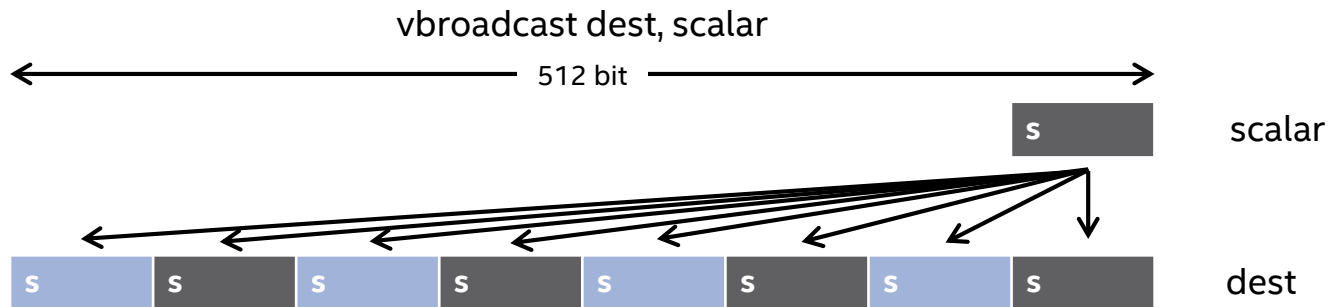
SIMD Instructions – Conditional Evaluation

Mask register limit effect of instructions to a subset of the SIMD elements



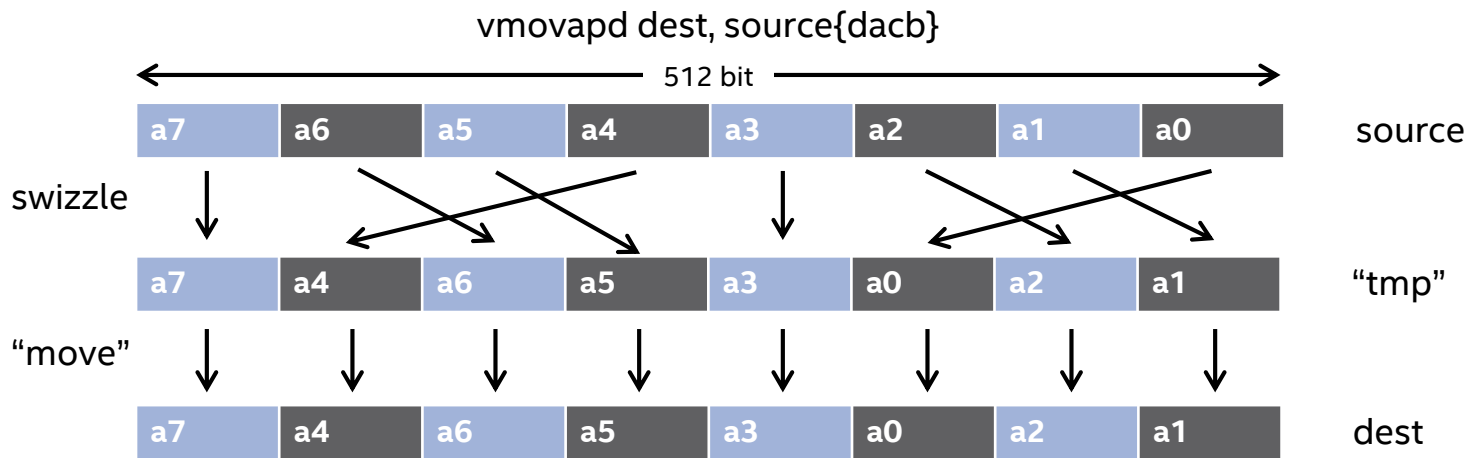
SIMD Instructions – Broadcast

Assign a scalar value to all SIMD elements

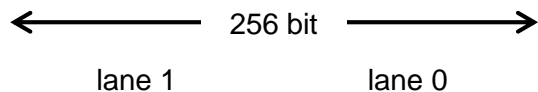


SIMD Instructions – Shuffles, Swizzles, Blends

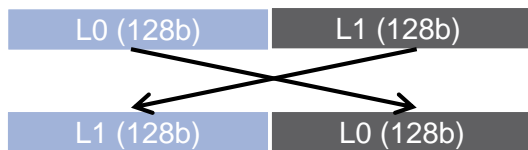
Instruction to modify data layout in the SIMD register



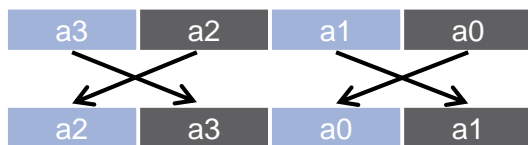
SIMD Instructions – Inter-lane Permutes



`vperm2f128 dest, src, src, 1`



`vpermilpd dest, src, 5`

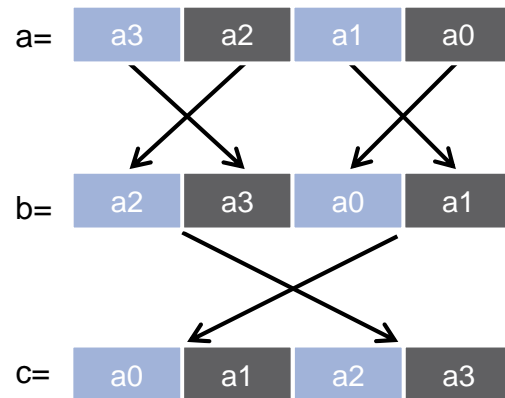


Example: register rotate

Immediate controls
permutation operation.

`vpermilpd b, a, 5`

`vperm2f128 c, b, b, 1`



SIMD INTRINSICS

SIMD through C Intrinsic Functions

Intrinsic functions

- Special functions recognized by the compiler
- Compiler attaches certain meaning to these functions
- Examples: `__builtin_prefetch()`, `__builtin_bswap32()`

SIMD intrinsic functions

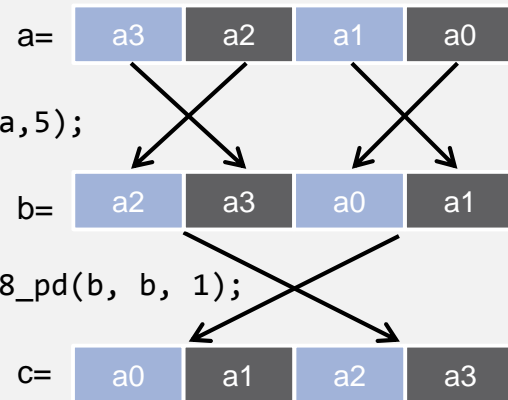
- Function correspond to sequence of one or more assembly instructions
- No correctness checking by the compiler (well mostly)
- Some optimizations are done, e.g., register assignment

```
#include <immintrin.h>
static inline
__m256d register_rotate_256d(__m256d a) {
    __m256d b, c;

    b = _mm256_permute_pd(a, 5);

    c = _mm256_permute2f128_pd(b, b, 1);

    return c;
}
```



Example: AVX Intrinsic Functions

AVX SIMD Data Types

- `__m256`: a vector of 8 float entries
- `__m256d`: a vector of 4 double entries
- `__m256i`: a vector of 4 longs (or 8 int, ...)

Intrinsic functions

`__m256 a = _mm256_add_pd(__m256 a, __m256 b)`

Vector length:

`_mm512`
`_mm256`
`_mm128`

Functionality:

- add
- mul
- sub
- load

...

Type and precision:

pd: packed double
ps: packed single
sd: scalar double
ss: scalar single

Intel® Intrinsic Guide

The screenshot shows the Intel Intrinsic Guide website in a web browser. The browser's address bar displays the URL <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>. The page features a search bar with the placeholder text `_mm_search`. On the left side, there are two sections: "Technologies" and "Categories".

Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☒ AVX
- ☒ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVM
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography

The main content area displays a list of intrinsics, each with its name, parameters, and a corresponding intrinsic name. The list includes:

- `__m256i _mm256_abs_epi16 (__m256i a)` `vpabsw`
- `__m256i _mm256_abs_epi32 (__m256i a)` `vpabsd`
- `__m256i _mm256_abs_epi8 (__m256i a)` `vpabsb`
- `__m256i _mm256_add_epi16 (__m256i a, __m256i b)` `vpaddw`
- `__m256i _mm256_add_epi32 (__m256i a, __m256i b)` `vpaddq`
- `__m256i _mm256_add_epi64 (__m256i a, __m256i b)` `vpaddq`
- `__m256i _mm256_add_epi8 (__m256i a, __m256i b)` `vpaddb`
- `__m256d _mm256_add_pd (__m256d a, __m256d b)` `vaddpd`
- `__m256 _mm256_add_ps (__m256 a, __m256 b)` `vaddps`
- `__m256i _mm256_adds_epi16 (__m256i a, __m256i b)` `vpaddsw`
- `__m256i _mm256_adds_epi8 (__m256i a, __m256i b)` `vpaddsb`
- `__m256i _mm256_adds_epu16 (__m256i a, __m256i b)` `vpaddusw`
- `__m256i _mm256_adds_epu8 (__m256i a, __m256i b)` `vpaddusb`
- `__m256d _mm256_addsub_pd (__m256d a, __m256d b)` `vaddsubpd`
- `__m256 _mm256_addsub_ps (__m256 a, __m256 b)` `vaddsubps`
- `__m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int count)` `vpalignr`
- `__m256d _mm256_and_pd (__m256d a, __m256d b)` `vandpd`
- `__m256 _mm256_and_ps (__m256 a, __m256 b)` `vandps`
- `__m256i _mm256_and_si256 (__m256i a, __m256i b)` `vpand`
- `__m256d _mm256_andnot_pd (__m256d a, __m256d b)` `vandnps`
- `__m256 _mm256_andnot_ps (__m256 a, __m256 b)` `vandnps`

Example: SIMDifying saxpy

Scalar code:

```
void saxpy(float* y, float* x,
          float a, int n) {
    for (int i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Scalar loop! Can we do better?

SIMD with intrinsic functions:

```
void saxpy_simd(float* y, float* x,
               float a, int n) {
    int ub = n - (n % 8);
    __m256 vy, vx, va, tmp;
    va = _mm256_set1_ps(a);
    for (int i = 0; i < ub; i += 8) {
        vy = _mm256_loadu_ps(&y[i]);
        vx = _mm256_loadu_ps(&x[i]);
        tmp = _mm256_mul_ps(va, vx);
        vy = _mm256_add_ps(tmp, vy);
        _mm256_storeu_ps(&y[i], vy);
    }
    for (int i = ub; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

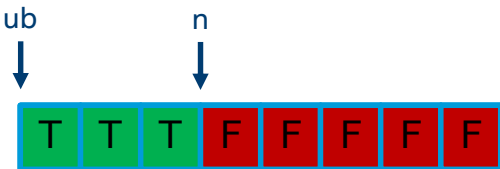
8 floats per SIMD register.

Example: SIMDifying saxpy – SIMD Remainder Loop

Remainder loop:

```
for (int i = ub; i < n; ++i) {  
    y[i] = a * x[i] + y[i];  
}
```

- Loop is needed to deal with iterations that are not a multiple of the SIMD length
- We cannot use a full vector, but we have mask instructions!



- TODO: Create a mask register with as many bits set as there are remaining iterations

Efficient *min* function¹ for int:

```
#define minint(x,y) (y^((x^y) & -(x < y)))
```

Mask creation:

```
__mmask8 mask;  
mask = (1 << (minint((ub+8), n) - ub)) - 1;
```

Alternative mask creation:

```
__mmask8 mask;  
mask = _bzhi_u32(0xFF, ub - n);
```

SIMD code for remainder loop:

```
vy = _mm256_mask_loadu_ps(vy, mask, &y[ub]);  
vx = _mm256_mask_loadu_ps(vx, mask, &x[ub]);  
tmp = _mm256_mask_mul_ps(va, mask, va, vx);  
vy = _mm256_mask_add_ps(vy, mask, tmp, vy);  
_mm256_mask_storeu_ps(&y[ub], mask, vy);
```

Example: SIMDifying saxpy

```
void saxpy_simd(float* y, float* x,
               float a, int n) {
    int ub = n - (n % 8);
    __m256 vy, vx, va, tmp;
    va = _mm256_set1_ps(a);
    for (int i = 0; i < ub; i += 8) {
        vy = _mm256_loadu_ps(&y[i]);
        vx = _mm256_loadu_ps(&x[i]);
        tmp = _mm256_mul_ps(va, vx);
        vy = _mm256_add_ps(tmp, vy);
        _mm256_storeu_ps(&y[i], vy);
    }

    // continue in right column
```

```
// continued from left column

    __mmask8 m;
    m = (1 << (minint(ub+8, n) - ub)) - 1;
    vy = _mm256_mask_loadu_ps(vy, m, &y[ub]);
    vx = _mm256_mask_loadu_ps(vx, m, &x[ub]);
    tmp = _mm256_mask_mul_ps(va, m, va, vx);
    vy = _mm256_mask_add_ps(vy, m, tmp, vy);
    _mm256_mask_storeu_ps(&y[ub], m, vy);
}
```

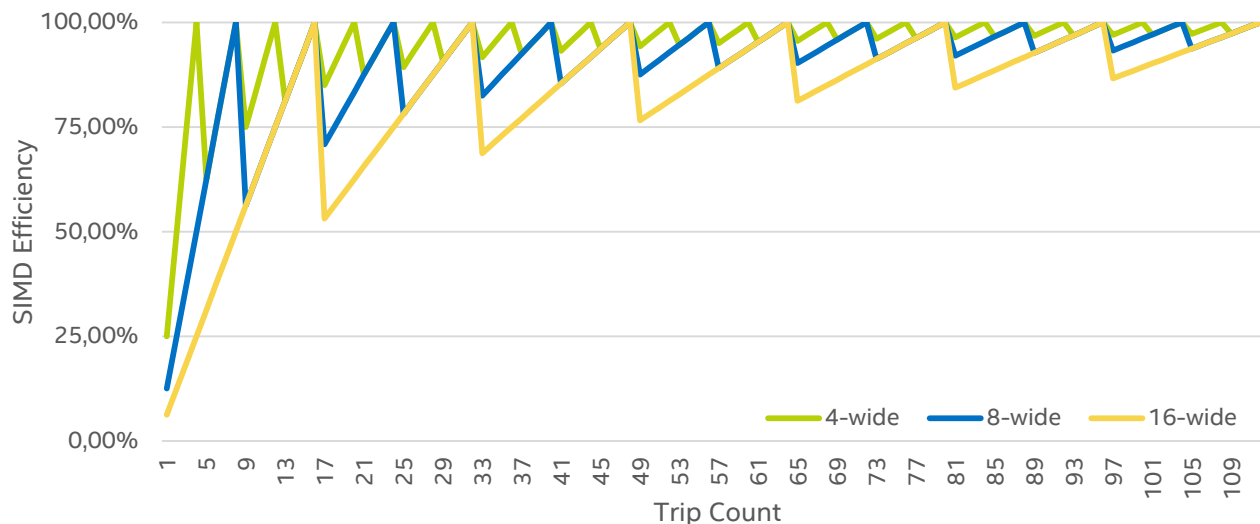
Vectorization Efficiency

Vectorization efficiency is a measure how well the code uses SIMD features

- Corresponds to the average utilization of SIMD registers for a loop
- Defined as (N : trip count, vl : vector length):
$$VE = \frac{N/vl}{\lceil N/vl \rceil}$$

For 8-wide SIMD:

- $N = 1$: 12.50%
- $N = 2$: 25.00%
- $N = 4$: 50.00%
- $N = 8$: 100.00%
- $N = 9$: 56.25%
- $N = 16$: 100.00%



Better Approach: High-level SIMD Programming

Programming with intrinsic functions is error-prone and less productive

- Only slightly higher level than assembly coding
- Moving to a different instruction set requires to rewrite (almost) everything

Better approach: compiler-generated SIMD code

Auto-vectorization

Compilers offer auto-vectorization as an optimization pass

- Usually part of the general loop optimization passes
- Code analysis detects code properties that inhibit SIMD vectorization
- Heuristics determine if SIMD execution might be beneficial
- If all goes well, the compiler will generate SIMD instructions



Example: Intel® Composer XE

- -vec (automatically enabled with -O2)
- -qopt-report

Interlude: Data Dependencies

Suppose two statements S1 and S2


S2 depends on S1, iff S1 must execute before S2

- Control-flow dependence
- Data dependence
- Dependencies can be carried over between loop iterations

Important flavors of data dependencies


FLOW

```
s1: a = 40
    b = 21
s2: c = a + 2
```



ANTI

```
    b = 40
s1: a = b + 1
s2: b = 21
```



Interlude: Loop-carried Dependencies

Dependencies may occur across loop iterations

- Then they are called “loop-carried dependencies”
- “Distance” of a dependency: number of loop iterations the dependency spans

The following code contains such a dependency:

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

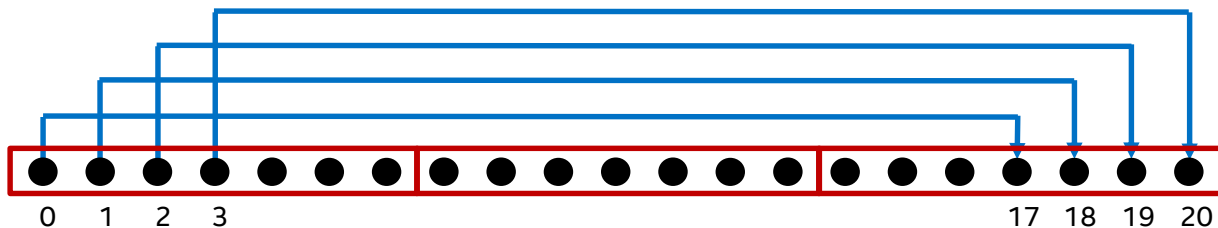
Loop-carried dependency for
a[i] and a[i+17]; distance is 17.

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?
- Note: This condition is sufficient, but not necessary!

Interlude: Loop-carried Dependencies

Can we parallelize or vectorize the loop?



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

- Parallelization: no
(except for very specific loop schedules)
- Vectorization: yes
(iff vector length is shorter than any distance of any dependency)

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


Fortran Array Notations

Fortran offers special syntax to execute operations on an array

- Simply coding: no need to write loops
- Less and more concise code

Example:

```
module daxpy_mod
contains
  subroutine daxpy(y, x, a)
    use iso_fortran_env
    real(kind=real64), intent(inout) :: y(:)
    real(kind=real64), intent(in) :: x(:)
    real(kind=real64), intent(in) :: a
    y = a * x + y
  end subroutine
end module
```



```
do i=1,size(y)
  y(i) = a * x(i) + y(i)
end do
```

OPENMP* SIMD PROGRAMMING

*Other names and brands may be claimed as the property of others.

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++)

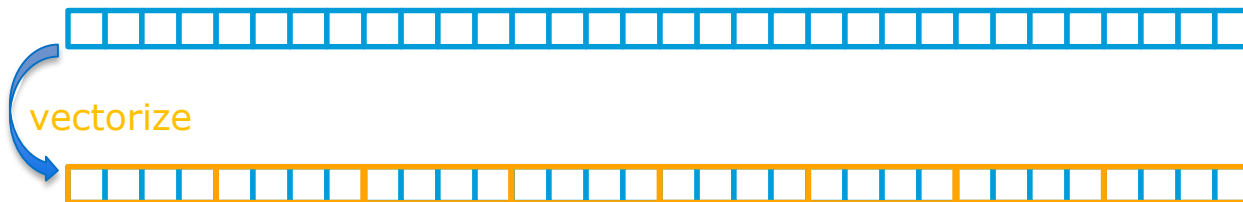
```
#pragma omp simd [clause[[, clause],...]  
for-loops
```

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
 - $x_i = x_{\text{orig}} + i * \text{linear-step}$

`aligned (list[:alignment])`

- Specifies that the list items have a given alignment
- Default is alignment for the architecture

`collapse (n)`

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)

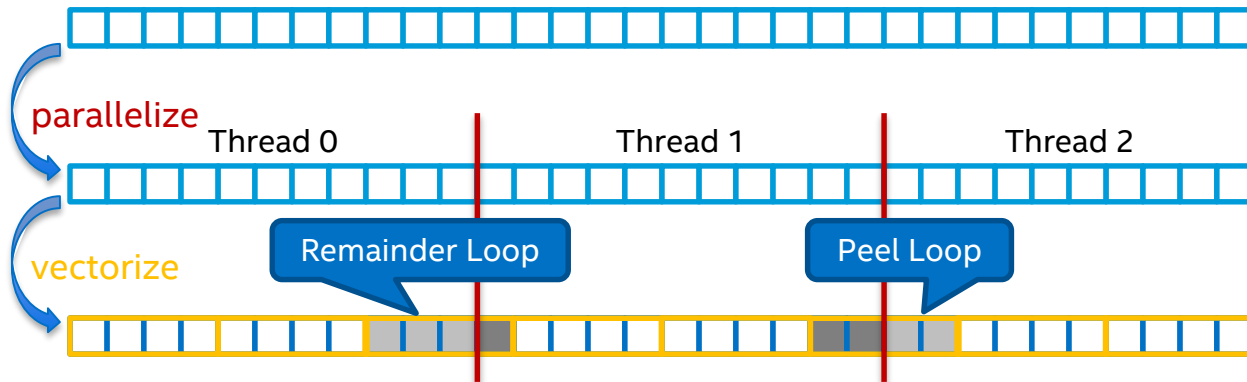
```
!$omp do simd [clause[[,] clause],...]
```

do-Loops

```
[!$omp end do simd [nowait]]
```

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 (= 8-wide), the code will only execute the remainder loop!
- and SSE (=4-wide), the code will have one iteration in the SIMD loop plus one in the remainder loop!

OpenMP 4.5 SIMD Chunks

```
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;  
}
```

Chooses chunk sizes that are multiples of the SIMD length

- First and last chunk may be slightly different to fix alignment and to handle loops that are not exact multiples of SIMD width
- Remainder loops are not triggered
- Likely better performance

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;  
}
```



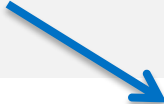
```
_ZGVZN16vv_min(%zmm0, %zmm1):  
    vminps %zmm1, %zmm0, %zmm0  
    ret
```

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



```
_ZGVZN16vv_distsq(%zmm0, %zmm1):  
    vsubps %zmm0, %zmm1, %zmm2  
    vmulps %zmm2, %zmm2, %zmm0  
    ret
```

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



```
vmovups (%r14,%r12,4), %zmm0  
vmovups (%r13,%r12,4), %zmm1  
call _ZGVZN16vv_distsq  
vmovups (%rbx,%r12,4), %zmm1  
call _ZGVZN16vv_min
```

AT&T syntax: destination operand is on the right

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`

- optimize for 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]*)

SIMD CONSIDERATIONS

Data Layout – Why It's Important

Instruction-Level

- Hardware is optimized for contiguous loads/stores.
- Support for non-contiguous accesses differs with hardware.
(e.g., AVX2 gather vs. AVX-512 gather/scatter)

Memory-Level

- Contiguous memory accesses are cache-friendly.
- Number of memory streams can place pressure on prefetchers.

Data Layout – Common Layouts

Array-of-Structs (AoS)

x	y	z	x	y	z
x	y	z	x	y	z
x	y	z	x	y	z

Pros:
Good locality of {x, y, z}.
1 memory stream.

Cons:
Potential for gather &
scatter operations

Struct-of-Arrays (SoA)

x	x	x	x	x	x
y	y	y	y	y	y
z	z	z	z	z	z

Pros:
Contiguous load/store.

Cons:
Poor locality of {x, y, z}.
3 memory streams.

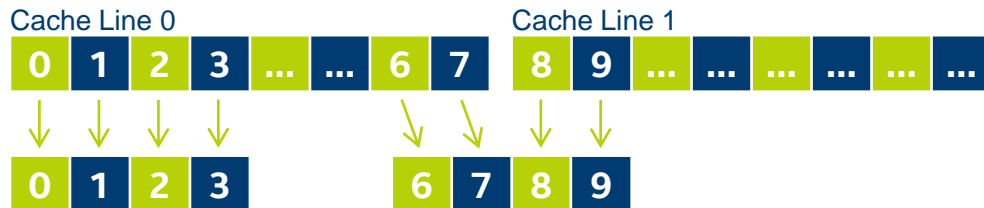
Hybrid (AoSoA)

x	x	y	y	z	z
x	x	y	y	z	z
x	x	y	y	z	z

Pros:
Contiguous load/store.
1 memory stream.

Cons:
Not a “normal” layout.

Data Alignment – Why It's Important



Aligned Load

- Address is aligned.
- One cache line.
- One instruction.

Unaligned Load

- Address is not aligned.
- Potentially multiple cache lines.
- Potentially multiple instructions.

Controlling Data Alignment

Align memory

- `_mm_malloc(bytes, 64)` / `!dir$ attributes align:64`

Access Memory in an aligned way

- `for (i = 0; i < N; i++) { array[i] ... }`

Tell the compiler!

- `#pragma omp simd aligned(...)` / `!$omp simd aligned(...)`
- `__assume_aligned(p, 16)` / `!dir$ assume_aligned (p, 16)`
- `__assume(i % 16 == 0)` / `!dir$ assume (mod(i, 16) .eq. 0)`

Example: saxpy – Improper Alignment

```
void saxpy(float* y, float* x,  
          float a, int n) {  
    #pragma omp simd  
    for (int i = 0; i < n; ++i) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

Resulting code on Intel Xeon Phi Coprocessors

```
vloadunpackld (%rsi,%r14,4),%zmm1  
vprefetch1 0x400(%rsi,%r14,4)  
vloadunpackld 0x40(%rsi,%r14,4),%zmm2  
vprefetch0 0x200(%rsi,%r14,4)  
vloadunpackhd 0x40(%rsi,%r14,4),%zmm1  
vprefetche1 0x400(%rdi,%r14,4)  
vloadunpackhd 0x80(%rsi,%r14,4),%zmm2  
vprefetch0 0x200(%rdi,%r14,4)  
vfmadd213ps (%rdi,%r14,4),%zmm0,%zmm1  
vprefetch1 0x440(%rsi,%r14,4)  
vfmadd213ps 0x40(%rdi,%r14,4),%zmm0,%zmm2  
vprefetch0 0x240(%rsi,%r14,4)  
vmovaps %zmm1, (%rdi,%r14,4)  
vprefetche1 0x440(%rdi,%r14,4)  
vmovaps %zmm2, 0x40(%rdi,%r14,4)  
vprefetch0 0x240(%rdi,%r14,4)
```

Example: saxpy – Improper Alignment

```
void saxpy(float* y, float* x,  
          float a, int n) {  
#pragma omp simd aligned(x,y:64)  
    for (int i = 0; i < n; ++i) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

Resulting code on Intel Xeon Phi Coprocessors

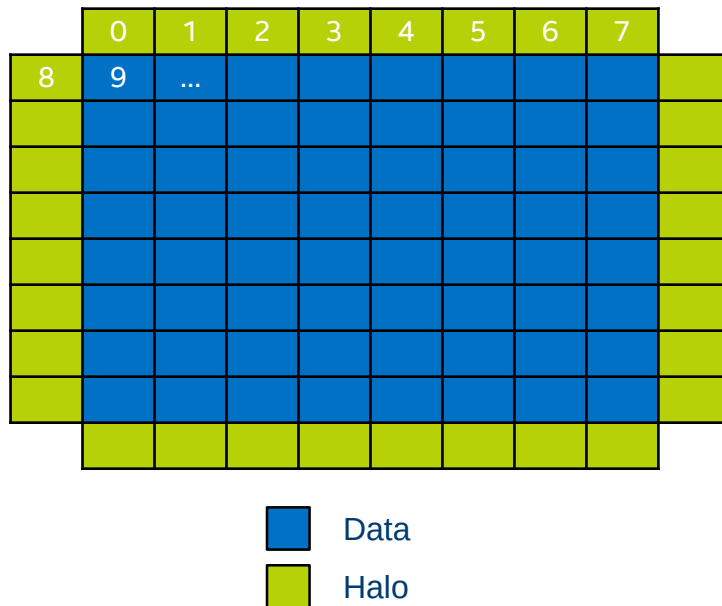
```
vmovaps (%rsi,%r9,4),%zmm2  
vprefetch1 0x400(%rsi,%r9,4)  
vmovaps 0x40(%rsi,%r9,4),%zmm3  
vprefetch0 0x200(%rsi,%r9,4)  
vfmadd213ps (%rdi,%r9,4),%zmm0,%zmm2  
vprefetch1 0x400(%rdi,%r9,4)  
vfmadd213ps 0x40(%rdi,%r9,4),%zmm0,%zmm3  
vprefetch0 0x200(%rdi,%r9,4)  
vmovaps %zmm2, (%rdi,%r9,4)  
vprefetch1 0x440(%rsi,%r9,4)  
vmovaps %zmm3, 0x40(%rdi,%r9,4)  
vprefetch0 0x240(%rsi,%r9,4)  
vprefetch1 0x440(%rdi,%r9,4)  
mov %al,%al  
vprefetch0 0x240(%rdi,%r9,4)
```

Data Alignment – n-dim Arrays

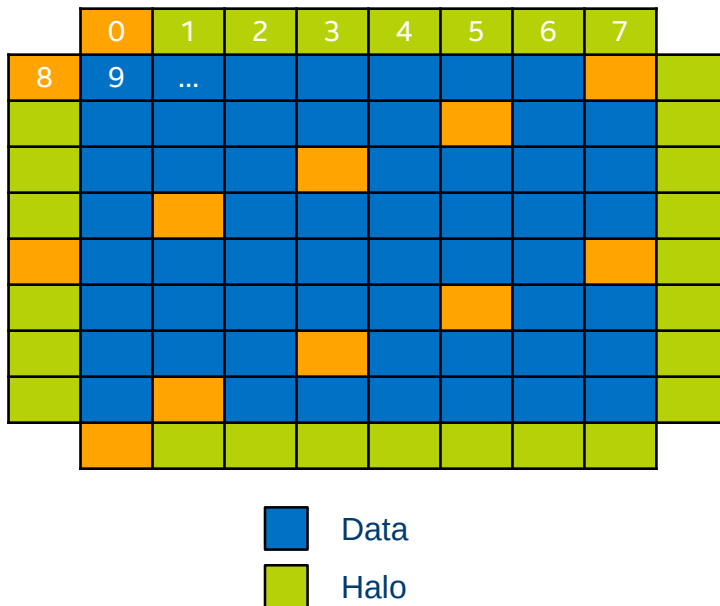
0	1	2	3	4	5	6	7
8	9	...					

 Data

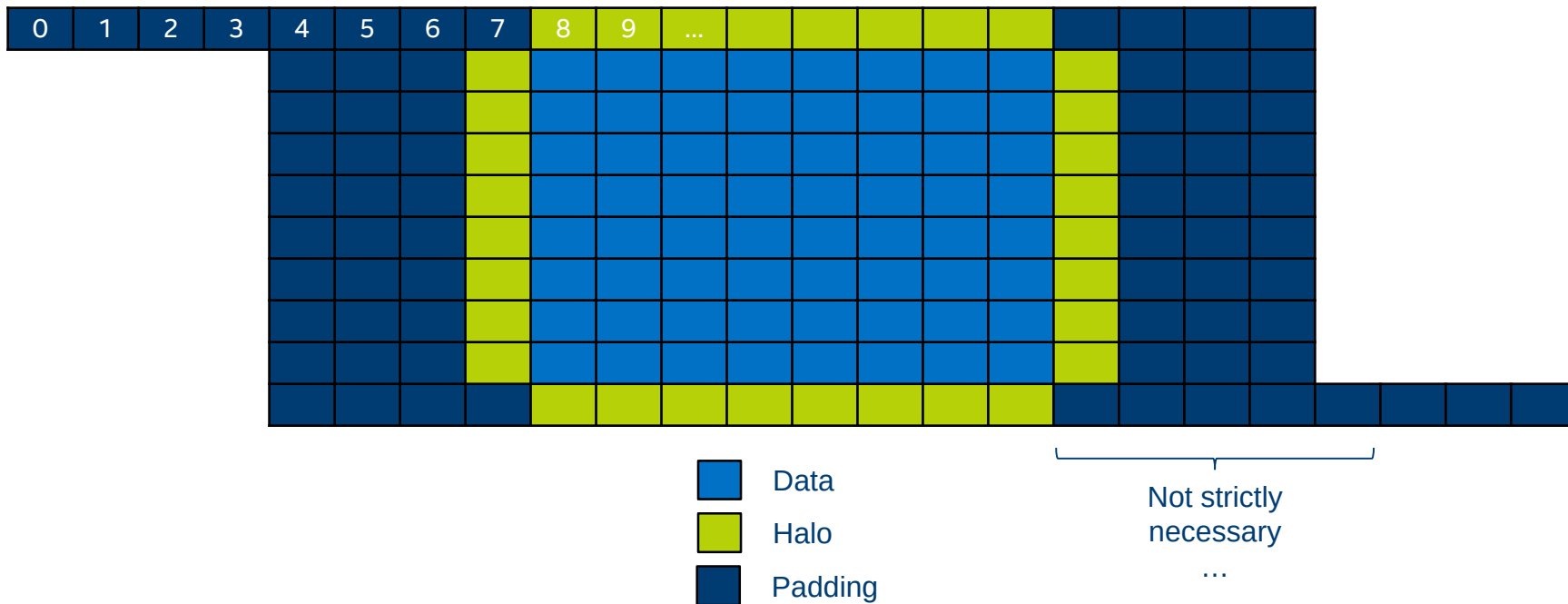
Data Alignment – n-dim Arrays



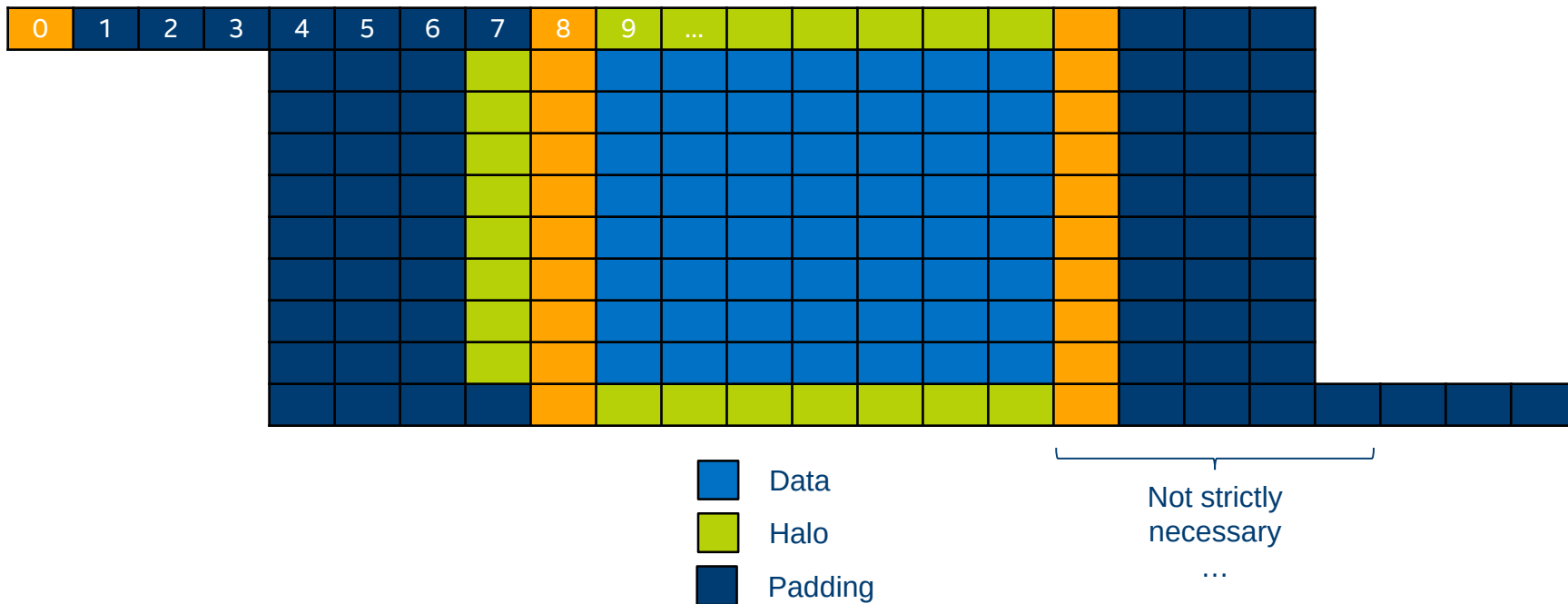
Data Alignment – n-dim Arrays



Data Alignment – n-dim Arrays



Data Alignment – n-dim Arrays



SUMMARY

Summary

Single-Instruction Multiple-Data (SIMD)

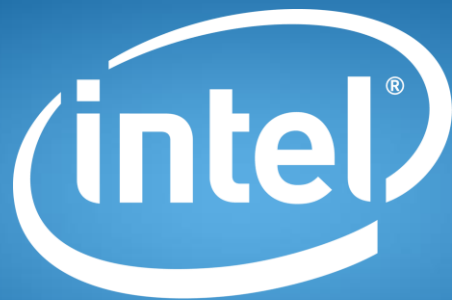
- Instruction-level parallelism
- Multiple data elements are processed in a single instruction
- Exploiting SIMD through low-level programming is cumbersome

OpenMP SIMD

- Easy access to the SIMD features of a processor
- Programmers help the compiler infer SIMD nature of the code

Efficient SIMD may requires special data structures

- Proper alignment for more efficient execution
- AoS to SoA conversion for improved memory access



experience
what's inside™