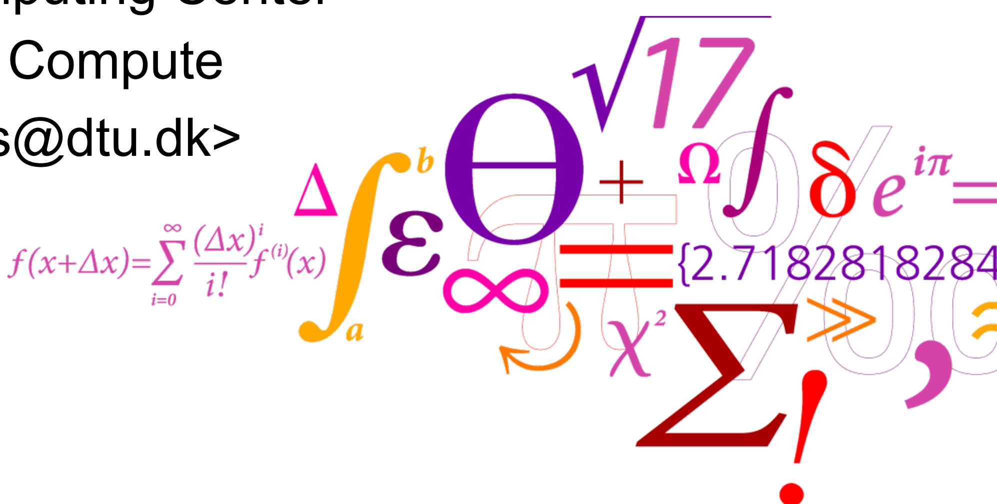
Vectorization in OpenMP 4.x

Hans Henrik Brandenborg Sørensen
DTU Computing Center
DTU Compute
<hhbs@dtu.dk>



Overview

- Introduction to vectorization
  - What is vectorization?
  - How to vectorize?
  - Data alignment
  - Data dependencies
- OpenMP 4.x vectorization
  - SIMD construct
  - Declare SIMD construct
  - New clauses
  - Examples

What is vectorization?

- Consider adding two vectors (1D arrays)

a:	3.1	1.2	7.7	6.0	0.3	4.2	8.9	1.2	2.1	1.0
+ b:	1.4	5.1	1.7	3.2	8.2	4.4	0.9	2.2	6.1	2.7
= c:	4.5	6.3	9.4	9.2	8.5	8.6	9.8	3.4	8.2	3.7

What is vectorization?

- Consider adding two vectors (1D arrays)

	i=0	i=1	i=2	i=3	…					
a:	3.1	1.2	7.7	6.0	0.3	4.2	8.9	1.2	2.1	1.0
+ b:	1.4	5.1	1.7	3.2	8.2	4.4	0.9	2.2	6.1	2.7
= c:	4.5	6.3	9.4	9.2	8.5	8.6	9.8	3.4	8.2	3.7

```c
#define N 10
double a[N], b[N], c[N];
…
for(int i=0; i<N; i++) {
   c[i] = a[i]+b[i];
}
```

What is vectorization?

- Consider adding two vectors (1D arrays)

	i=0	i=1	i=2	i=3	…					
a:	3.1	1.2	7.7	6.0	0.3	4.2	8.9	1.2	2.1	1.0
+ b:	1.4	5.1	1.7	3.2	8.2	4.4	0.9	2.2	6.1	2.7
= c:	4.5	6.3	9.4	9.2	8.5	8.6	9.8	3.4	8.2	3.7

```c
#define N 10
double a[N], b[N], c[N];
…
for(int i=0; i<N; i++) {
    c[i] = a[i]+b[i];
}
```

gcc -O2

Scalar code

```
.L2:
 movsd  (%rdi,%rax), %xmm0
 addsd  (%rsi,%rax), %xmm0
 movsd  %xmm0, (%rdx,%rax)
 addq   $8, %rax
 cmpq   $80, %rax
 jne     .L2
```

What is vectorization?

- Consider adding two vectors (1D arrays)

	i=0				i=4				…	
a:	3.1	1.2	7.7	6.0	0.3	4.2	8.9	1.2	2.1	1.0
+ b:	1.4	5.1	1.7	3.2	8.2	4.4	0.9	2.2	6.1	2.7
= c:	4.5	6.3	9.4	9.2	8.5	8.6	9.8	3.4	8.2	3.7

```c
#define N 10
double a[N], b[N], c[N];
…
for(int i=0; i<N; i++) {
    c[i] = a[i]+b[i];
}
```

`gcc -O2` → `-ftree-vectorize -mavx`

Vectorized / packed code

```
.L2:
 vmovapd (%rsi,%rax),%ymm0
 vaddpd (%rdi,%rax),%ymm0,%ymm0
 vmovapd %ymm0,(%rdx,%rax)
 addq   $32, %rax
 cmpq   $80, %rax
 jne     .L2
```

Terminology

- Vectorization is the process of transforming a scalar operation that acts on a single data element at a time (Single Instruction Single Data – SISD) to an operation that that acts on multiple data elements at a time (Single Instruction Multiple Data – SIMD).

- SIMD units are hardware arithmetic vector units that can perform the same operation on multiple data points simultaneously by using vector registers.

Vector lengths

- 128 bit: SSE = Streaming SIMD Extension (1999)

2 × double

4 × float

- 256 bit: AVX = Advanced Vector Extension (2011)

4 × double

8 × float

- 512 bit: AVX-512 (2013)

8 × double

16 × float

Vector impact grows

- Simpler circuit design
  - Costs less to simple widen the execution units
- More energy efficient with fewer instructions
  - 4-16 times fewer instructions are decoded and issued
- Memory access with known patterns are good
- Vector lenghts are likely to increase in the future – seems to double every four years

Ways to vectorize

- Compiler / explicit pragmas:
  - Auto-vectorization (no change of code)
  - Auto-vectorization hints (#pragma vector, …)
  - OpenMP 4.x (#pragma omp simd …)
- Low-level vector programming:
  - C++ intrinsic classes (e.g.: `F32vec`, `F64vec`, …)
  - Vector intrinsics (e.g.: `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)
  - Assembler code (e.g.: `[v]addps`, `[v]addss`, …)

➢ Ease of use

➢ Programmer control

Why always prefer the compiler?

- Easier and more readable code
- Portable across vendors and machines
  - Although compiler directives differ across compilers
- Better performance of the compiler generated code compared to explicit vector programmer
  - Compiler applies other transformations as well

However, we may need to help the compiler:

- Programmers may need to provide the necessary information (alignment, aliasing, inline)
- Programmers may need to transform the code

Compiler options

- Enabling with `gcc` (6.x): `-O2 -ftree-vectorize`
  - Specifying vector length: `-msse |-mavx | -mavx2`
  - For Haswell instructions: `-mfma -mavx2`
  - Using OpenMP 4.x vectorization: `-fopenmp-simd`
- Enabling with `icc` (17.x): `-O2`
  - Specifying vector length: `-novec |-msse |-mavx`
  - For Haswell instructions: `-fma -march=core-avx2`
  - Using OpenMP 4.x vectorization: `-fopenmp`
- Enabling with Sun Studio `cc` (12.x): `-xvector`

Vectorization reports

- Reports from `gcc` (6.x) to `stdout`
  - Specifying success: `-fopt-info-vec-optimized`
  - Specifying failure: `-fopt-info-vec-missed`
- Reports from `icc` (17.x) to file `<file>.optrpt`
  - `-qopt-report=5 -qopt-report-phase=vec`
  - Specifying stdout: `-qopt-report-file=stdout`

```
remark #15300: LOOP WAS VECTORIZED
remark #15442: entire loop may be executed in remainder
remark #15450: unmasked unaligned unit stride loads: 2
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 8
remark #15477: vector cost: 2.000
remark #15478: estimated potential speedup: 3.400
remark #15488: --- end vector cost summary ---
```

Data alignment

- Good alignment

Address:	0	8	16	24	32	40	48	56
Data:	a[0]	a[1]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
Vectors:								

Data alignment

- Good alignment

Address:	0	8	16	24	32	40	48	56
Data:	a[0]	a[1]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
Vectors:								

- Bad alignment

Address:	0	8	16	24	32	40	48	56	64
Data:		a[0]	a[1]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
Vectors:									

Data alignment

- Good alignment

Address:	0	8	16	24	32	40	48	56
Data:	a[0]	a[1]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
Vectors:								

- Bad alignment

Address:	0	8	16	24	32	40	48	56	64
Data:		a[0]	a[1]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
Vectors:									

- Very bad alignment

Address:	0	4	12	20	28	36	44	52	60
Data:		a[0]	a[1]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
Vectors:									

Data alignment

- Data alignment is important to assist vectorization
  - Proper alignment allows `vmovapd`, etc
  - Unaligned accesses are costly (extra cache line, shifts)
  - Loop only vectorized if deemed efficient by the compiler
- Static variable declarations
  - `__attribute__((aligned(n))) var_name`
- Dynamic variable declarations
  - `_mm_malloc(SIZE, n)   (#include <immintrin.h>)`
  - `posix_memaligned(void **p, n, NUMBER)`

Example

- Vector add (N=1000, repeated 10^7 times)

a=aligned, u=unaligned

Alignment	`vmovapd, …`	`vmovupd, …`	`movsd, …`
Good	1.62 sec	2.76 sec	5.32 sec
Bad	Segmentation fault	3.12 sec	5.32 sec
Very bad	Segmentation fault	3.12 sec	5.54 sec

Haswell

Restricting pointer aliasing

- Aliasing prevents code from vectorizing optimally

```
void vecadd(double *a,
            double *b,
            double *c,
            int n)
{
  for(int i=0; i<n; i++)
     c[i] = a[i]+b[i];
}
```

Restricting pointer aliasing

- Aliasing prevents code from vectorizing optimally

```
void vecadd(double *a,
            double *b,
            double *c,
            int n)
{
  for(int i=0; i<n; i++)
     c[i] = a[i]+b[i];
}
```

- `gcc` compiler report:

Asserts whether the pointers overlap before entering loop

```
vecadd.cpp:6:23: note: loop vectorized
vecadd.cpp:6:23: note: loop versioned for
vectorization because of possible aliasing
```

Restricting pointer aliasing

- Use the C99 std `__restrict__` type qualifier

```
void vecadd(double * __restrict__ a,
            double * __restrict__ b,
            double * __restrict__ c,
            int n)
{
  for(int i=0; i<n; i++)
     c[i] = a[i]+b[i];
}
```

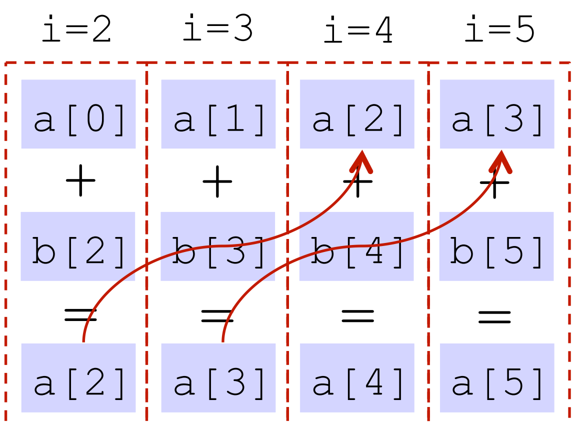
- Or use OpenMP 4.x (➔"I know what I'm doing!")
- `gcc` compiler report:

```
vecadd.cpp:6:23: note: loop vectorized
```

Data dependencies

- Loop carried data dependencies

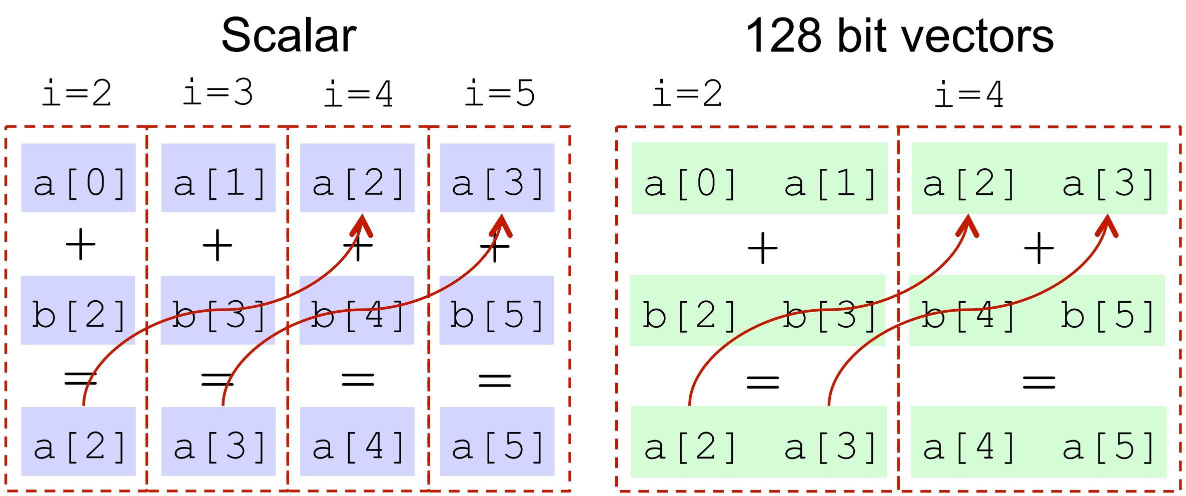
Scalar




```
for(int i=2; i<N; i++) {
   a[i] = a[i-2] + b[i];
}
```

Data dependencies

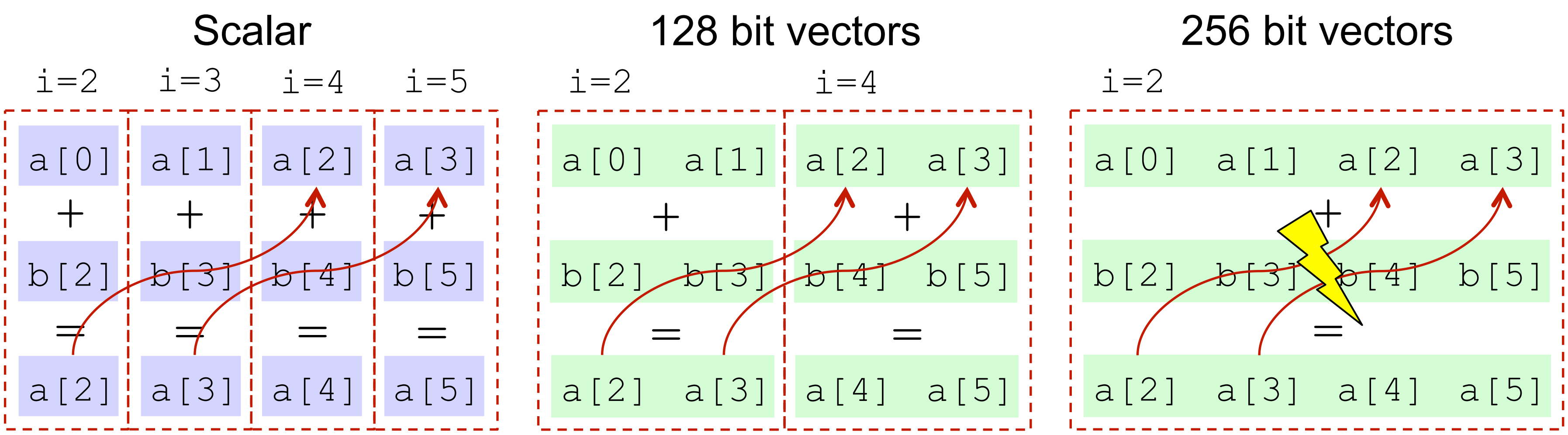
- Loop carried data dependencies




```
for(int i=2; i<N; i++) {
    a[i] = a[i-2] + b[i];
}
```

Data dependencies

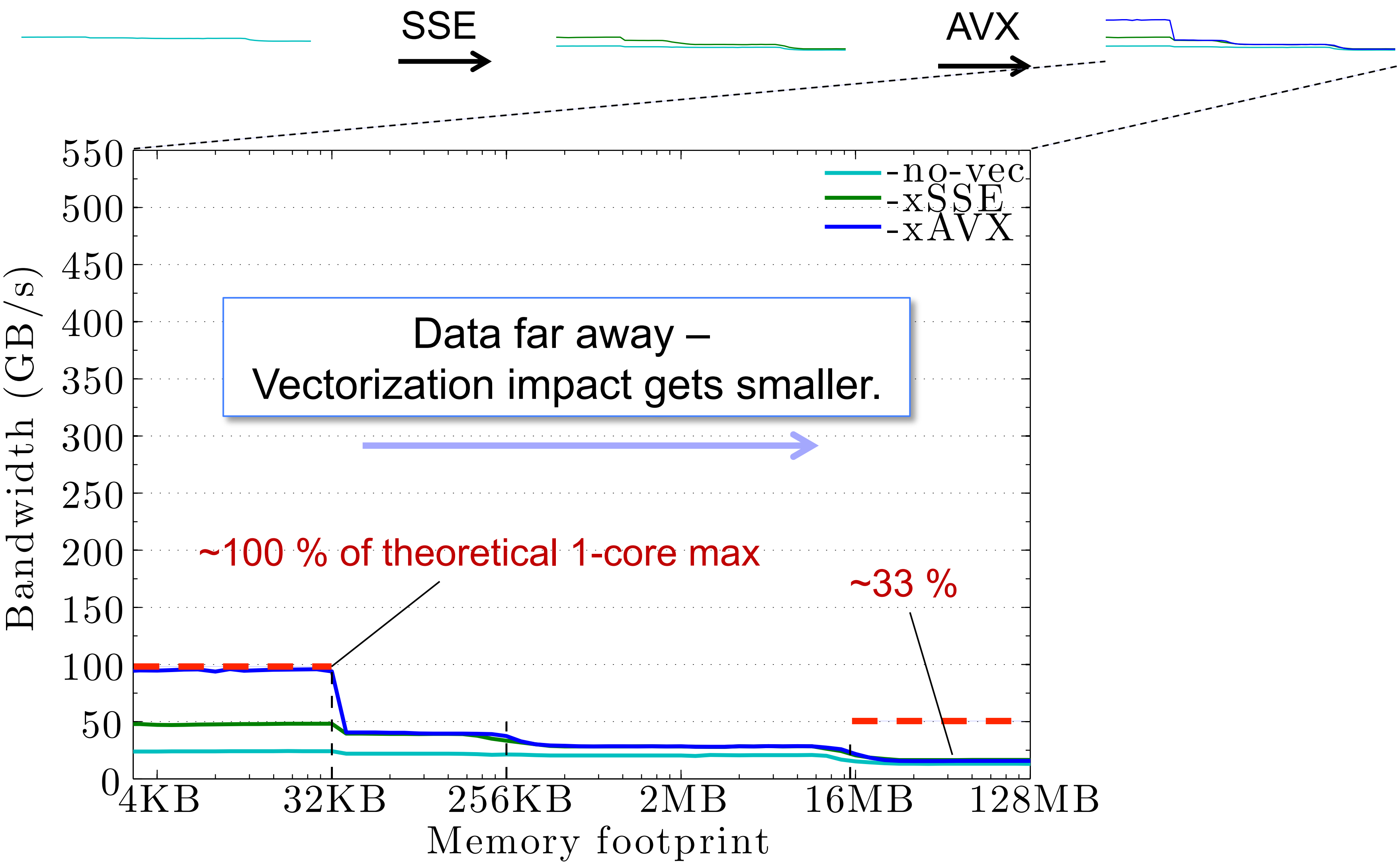
- Loop carried data dependencies




```
for(int i=2; i<N; i++) {
   a[i] = a[i-2] + b[i];
}
```

Loop carried / flow dependencies that cannot be disproven by to compiler prevents vectorization

Memory access



OpenMP 4.x SIMD syntax

OpenMP SIMD construct

- The basis of OpenMP 4.x vectorization is the `simd` construct / directive for vectorizing loops
- C/C++:

```
#pragma omp simd [clause[[,] clause],...]
for (...)
{...code block...}
```

- Fortran:

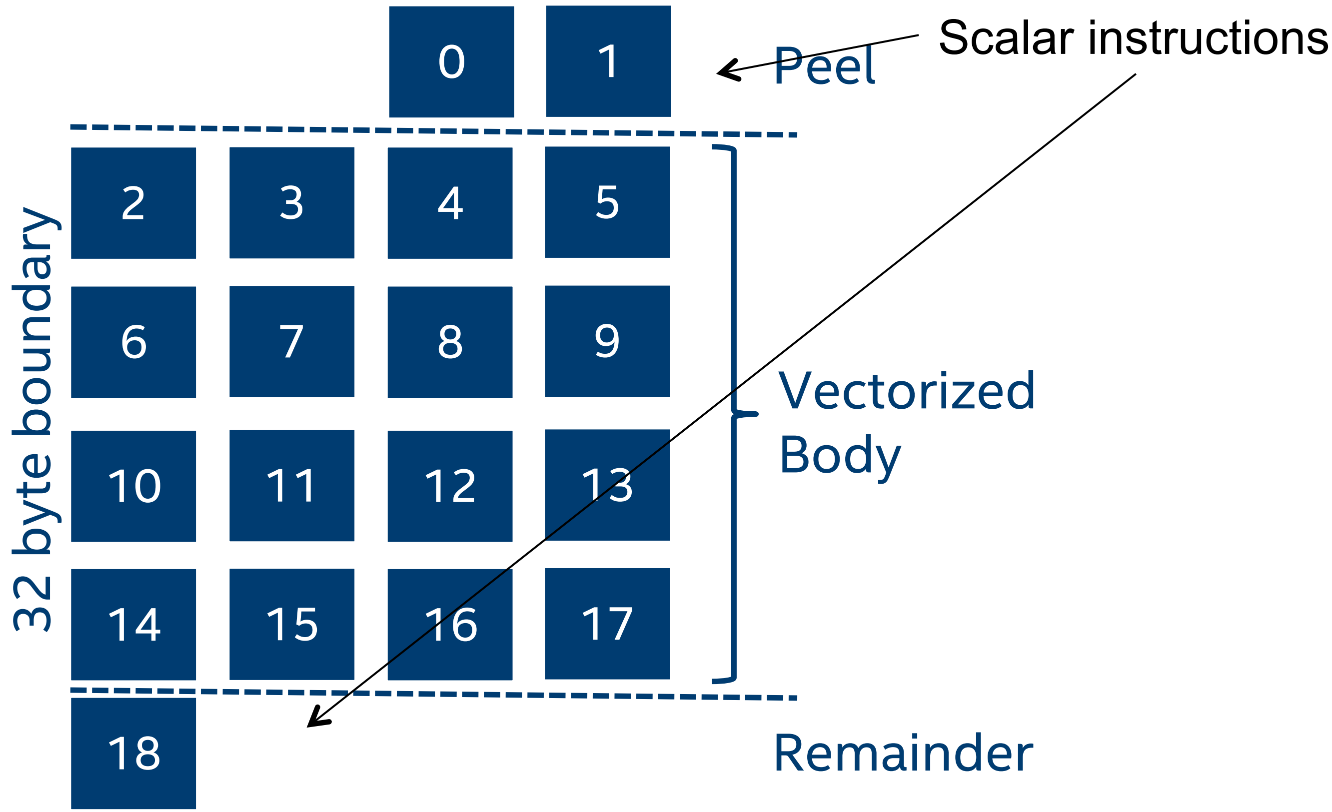
```
!$OMP simd [clause[[,] clause],...]
do-loops
!$OMP end simd
```

OpenMP SIMD construct

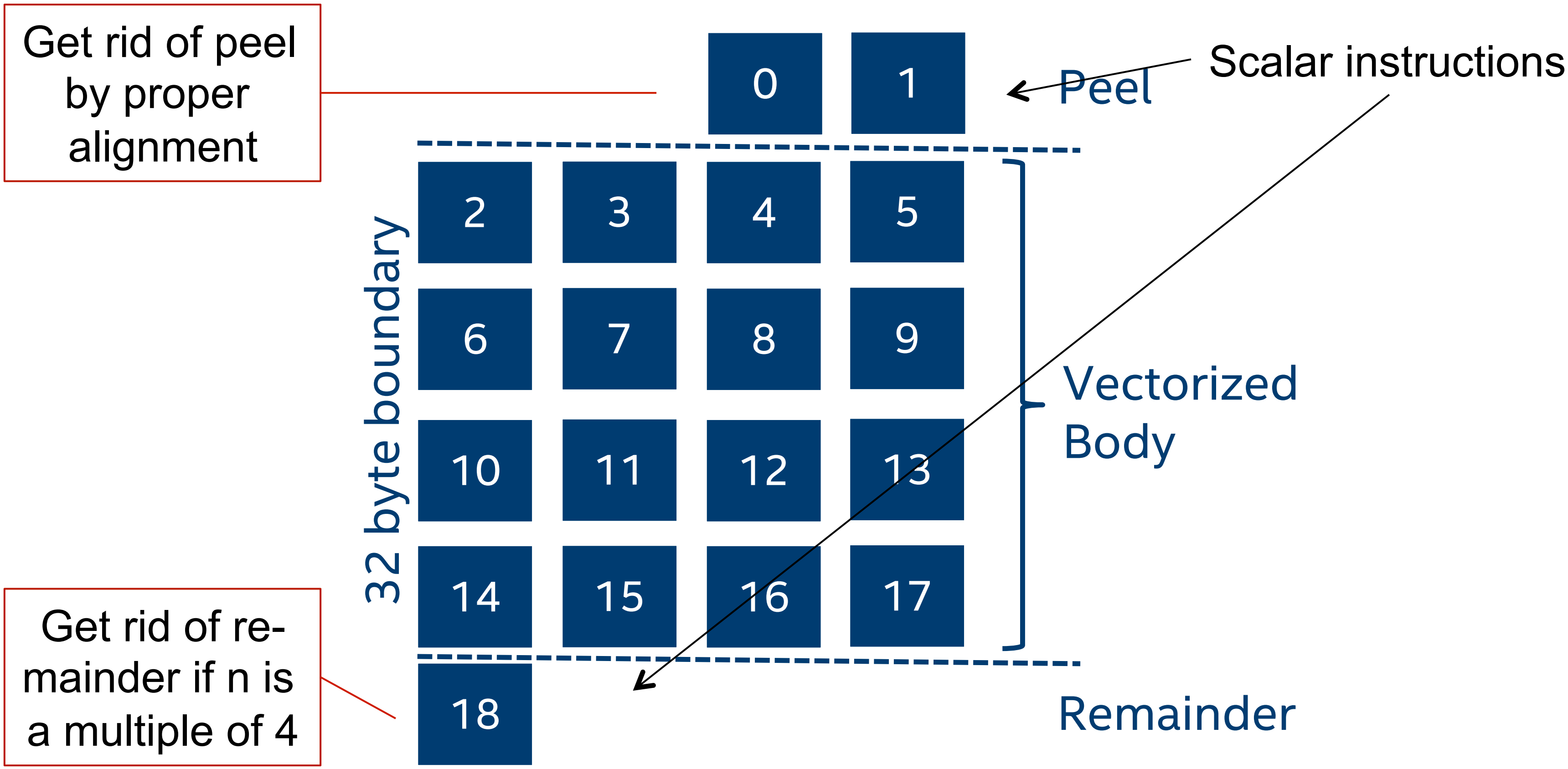
- Clauses
  - `private(list)`, as usual
  - `lastprivate(list)`, as usual
  - `reduction(operation:list)`, as usual
  - `collapse(n)`, collapse loops before simd, as usual
  - `linear(list[:linear-step])`, a variable increases linearly in every loop iteration
  - `aligned(list[:alignment])`, specifies that data is aligned
  - `safelen(n)`, distance of loop iterations where no data dependence occurs

New

Peels and remainders



Peels and remainders



- Make sure you execute as many of the loop's iterations as possible inside the vectorized body

Example I

- Scalar product of two vectors

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

OpenMP 4.x	gcc
`no vectorization`	8.76 sec
`#pragma omp simd reduction(+:sum)`	2.43 sec

3.60x

Example I

- Scalar product of two vectors

```
void sprod(double *a, double *b, int n) {
   double sum = 0.0;
   #pragma omp simd reduction(+:sum) aligned(a,b:32)
   for (int i=0; i<n; ++i)
      sum += a[i] * b[i];
   return sum;
}
```

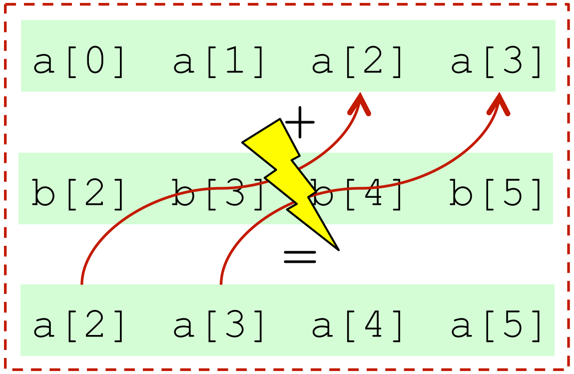
OpenMP 4.x	gcc
`no vectorization`	8.76 sec
`#pragma omp simd reduction(+:sum)`	2.43 sec
`#pragma omp simd reduction(+:sum) \ aligned(a,b:32)`	2.19 sec

4.00x

OpenMP SIMD safelen clause

- The `safelen(n)` clause is used to ensure that loop-carried dependencies are not violated
  - If specified, the loop may be safely vectorized with any vector length less than or equal to `n`

```
#pragma omp simd safelen(2)
for(int i=2; i<N; i++) {
    a[i] = a[i-2] + b[i];
}
```



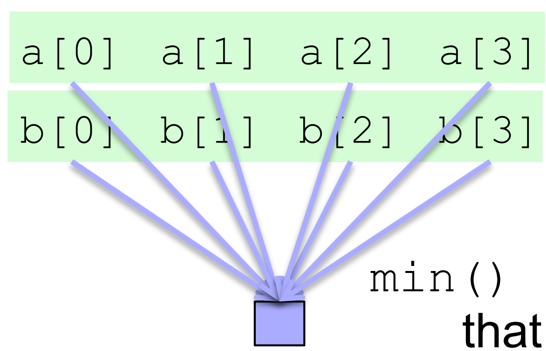
- Easiest way NOT to vectorize a loop in OpenMP?
  - Use `safelen(1)` clause on loop! (2017 only for `icc`)

Example II

- Min of two vectors

```
double min(double a, double b) {
    return a < b ? a : b;
}
```
min.cpp

```
void minvec(double *a, double *b, double *c, int n){
   #pragma omp simd aligned(a,b,c:32)
   for (int i=0; i<n; ++i)
       c[i] = min(a[i], b[i]);
}
```
main.cpp



`min()` is here a scalar function that is executed serially

OpenMP declare SIMD construct

- Declare functions that should be vectorized
  - Used for calls from within a SIMD loop
  - Is matched at compile time to call's vector length
- C/C++:

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

- Fortran:

```
!$OMP declare simd (proc-name) [clause[[,]
clause],...]
```

OpenMP declare SIMD construct

- Clauses
  - `reduction(operation:list)`, as usual
  - `linear(list[:linear-step])`, a variable increases linearly in every loop iteration
  - `aligned(list[:alignment])`, specifies that data is aligned
  - `simdlen(length)`, the vector length to be used
  - `uniform(list)`, arguments that have an invariant value in every loop iteration
  - `inbranch / notinbranch`, function is always/ never called from within a conditional statement

New

Example II

- Min of two vectors

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

min.cpp

```cpp
void minvec(double *a, double *b, double *c, int n){
   #pragma omp simd aligned(a,b,c:32)
   for (int i=0; i<n; ++i)
       c[i] = min(a[i], b[i]);
}
```

main.cpp

OpenMP 4.x	gcc
no vectorization	17.37 sec
#pragma omp simd aligned(a,b,c:32)	5.91 sec

2.93x

OpenMP worksharing construct

- OpenMP 4.x allows seamless integration of parallellization and vectorization
  - First; distribute loop's iterations over threads
  - Second; vectorize computation for each thread
- C/C++:

```
#pragma omp for simd [clause[[,] clause],...]
for (...)
{...code block...}
```

- Fortran:

```
!$OMP do simd [clause[[,] clause],...]
do-loops
!$OMP end simd
```

Example III

- Pi by integration

```
double f(double x) {
   return (4.0 / (1.0 + x*x));
}

double pi(long N) {
   double sum = 0.0, h = 1.0/N;
   #pragma omp parallel for reduction(+:sum)
   for (long i = 1; i <= N; i++) {
         double x = h * (i - 0.5);
         sum += f(x);
   }
   return h*sum;
}
```

Example III

- Pi by integration

```
#pragma omp declare simd
double f(double x) {
   return (4.0 / (1.0 + x*x));
}

double pi(long N) {
   double sum = 0.0, h = 1.0/N;
   #pragma omp parallel shared(sum)
   {
      #pragma omp for simd reduction(+:sum)
      for (long i = 1; i <= N; i++) {
         double x = h * (i - 0.5);
         sum += f(x);
      }
   }
   return h*sum; }
```

Example III

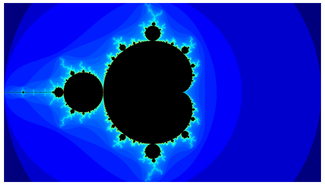
- Pi by integration

OpenMP 4.x	`OMP_NUM_THREADS`	`icc`
`no vectorization, #pragma omp parallel \ for reduction(+:sum)`	1	4.85 sec
	2	2.42 sec
	4	1.21 sec
	8	0.61 sec
	16	0.30 sec
`#pragma omp for \ simd reduction(+:sum)`	1	2.43 sec
	2	1.21 sec
	4	0.61 sec
	8	0.30 sec
	16	0.15 sec

32.3x

Example IV

- Mandelbrot 2048x2048



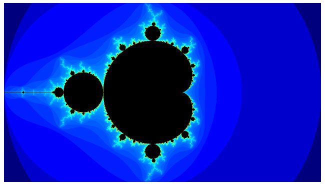
gcc

```
mandel.cpp:44:33: note: not vectorized: number of
iterations cannot be computed.
mandel.cpp:44:33: note: bad loop form.
```

icc

```
remark #15301: FUNCTION WAS VECTORIZED
[ mandel.cpp(42,1) ]
```

Example IV



- Mandelbrot 2048x2048

gcc
```
mandel.cpp:44:33: note: not vectorized: number of
iterations cannot be computed.
mandel.cpp:44:33: note: bad loop form.
```

icc
```
remark #15301: FUNCTION WAS VECTORIZED
[ mandel.cpp(42,1) ]
```

OpenMP 4.x / `NUM_THREADS=20`	`gcc`	`icc`
`no vectorization`	0.97 sec	0.83 sec
`#pragma omp declare simd \` `uniform(max_iter) simdlen(16)`	0.97 sec	0.17 sec

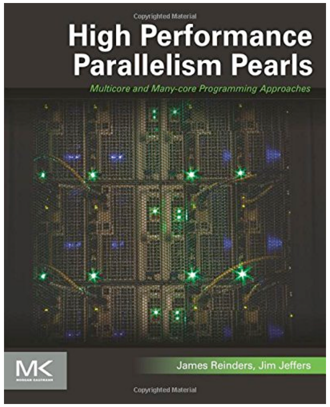
4.88x

GNU's compiler is currently not as mature as Intel's compiler when it comes to vectorizaton!

References

- Online material / documents:
  - Putting Vector Programming to Work With OpenMP SIMD, Intel Corp. *http://goparallel.sourceforge.net/wp-content/uploads/2015/09/ TheParallelUniverse_Issue_22-Feature1.pdf*
  - James Reinders presents: SIMD, Vectorization, and Performance Tuning. VIDEO: *http://insidehpc.com/2016/09/simd-vectorization-and-performance-tuning/*
  - *http://www.hpctoday.com/hpc-labs/explicit-vector-programming-with-openmp-4-0-simd-extensions/*
  - Wende *et al.,* Portable SIMD Performance with OpenMP* 4.x Compiler Directives, *http://link.springer.com/chapter/ 10.1007/978-3-319-43659-3_20*
- Useful book:
  - James Reinders, Jim Jeffers:

  "High Peformance Parallelism Pearls", 2015



End of lecture