Software

VECTORIZATION ESSENTIALS

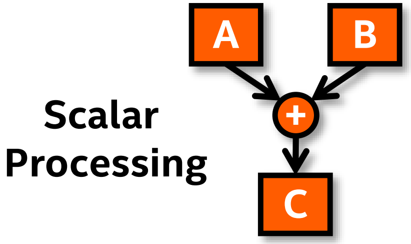
Agenda

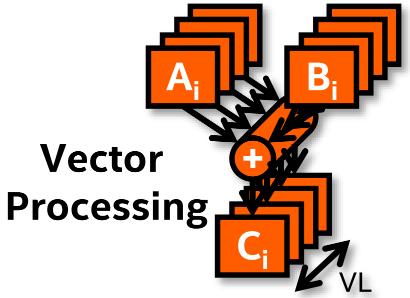
- **Introduction**
  - **Why Is Vectorization Important?**
  - **Basic Vectorization Terms**
  - **Evolution of SIMD for Intel® Processors**
- Auto-vectorization of Intel® Compilers
- Reasons for Vectorization Failures and Inefficiency

Single Instruction Multiple Data (SIMD)

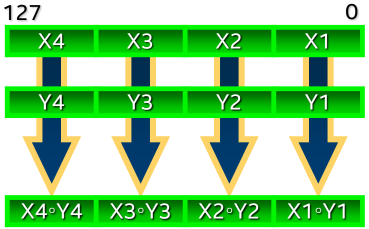
SIMD from Intel has been key for data level parallelism for years:

- **128 bit** Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3, SSE4.1, SSE4.2) and Supplemental Streaming SIMD Extensions (SSSE3)
- **256 bit** Intel® Advanced Vector Extensions (Intel® AVX)
- **512 bit** Intel® Advanced Vector Extensions 512 (Intel® AVX-512)





SIMD Types for Intel® Architecture



SSE

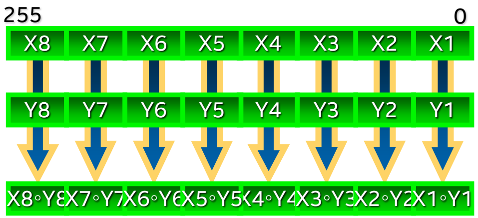
Vector size: **128 bit**

Data types:

8, 16, 32, 64 bit integer

32 and 64 bit float

VL: 2, 4, 8, 16



AVX

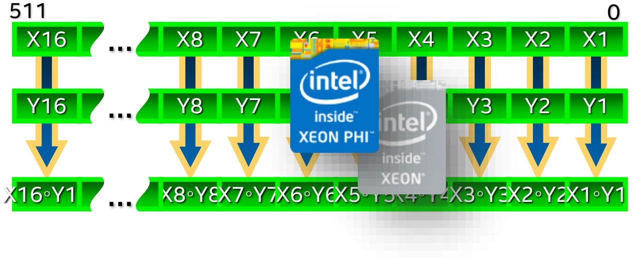
Vector size: **256 bit**

Data types:

8, 16, 32, 64 bit integer

32 and 64 bit float

VL: 4, 8, 16, 32



Intel® AVX-512

Vector size: **512 bit**

Data types:

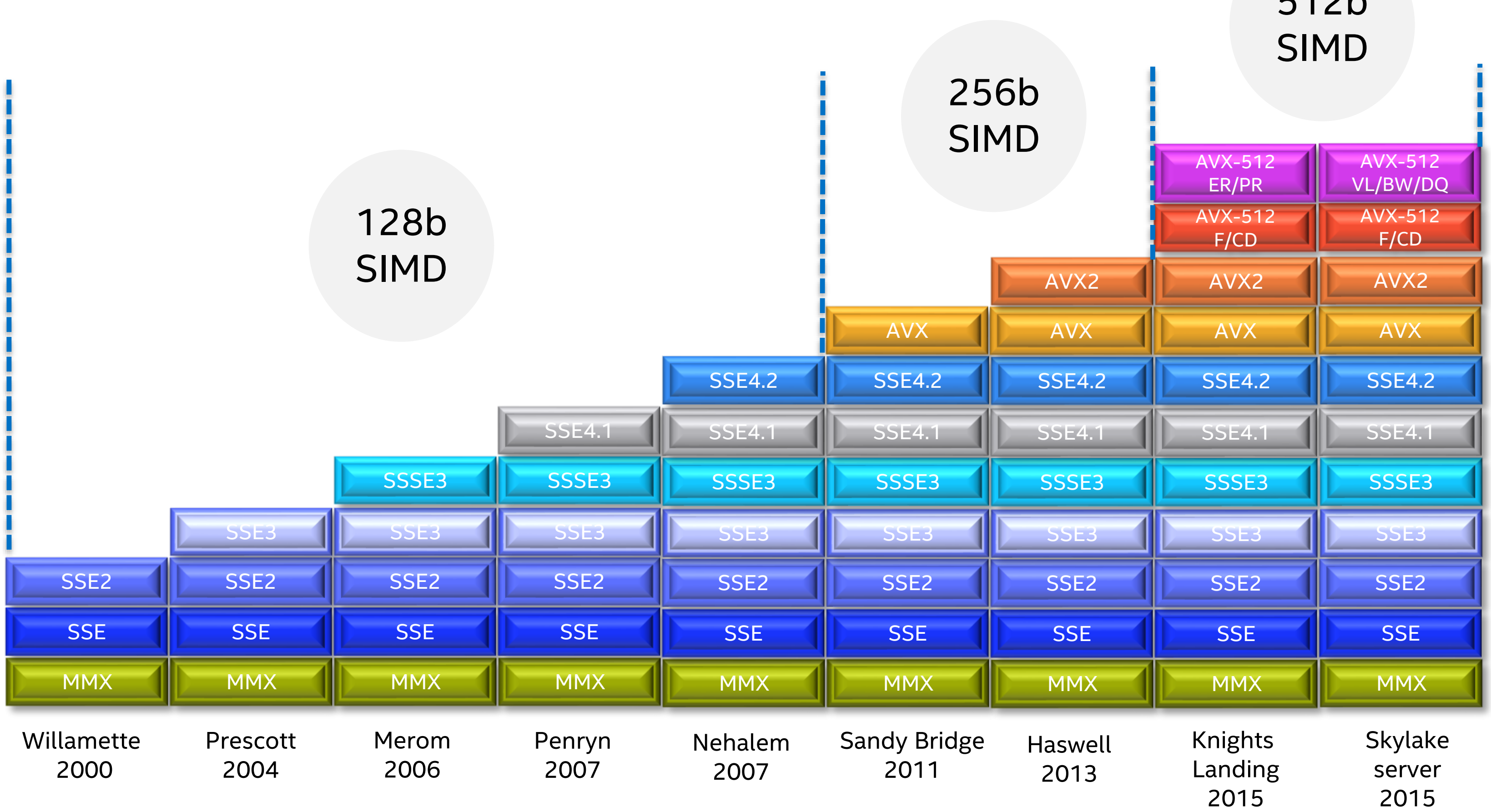
8, 16, 32, 64 bit integer

32 and 64 bit float

VL: 8, 16, 32, 64

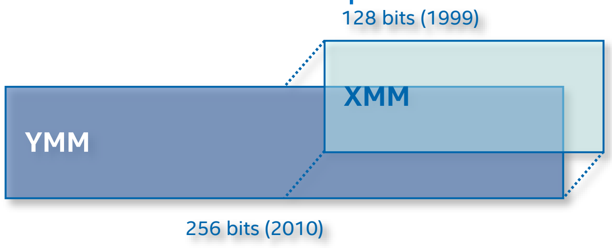
Illustrations: Xi, Yi & results 32 bit integer

Evolution of SIMD for Intel Processors

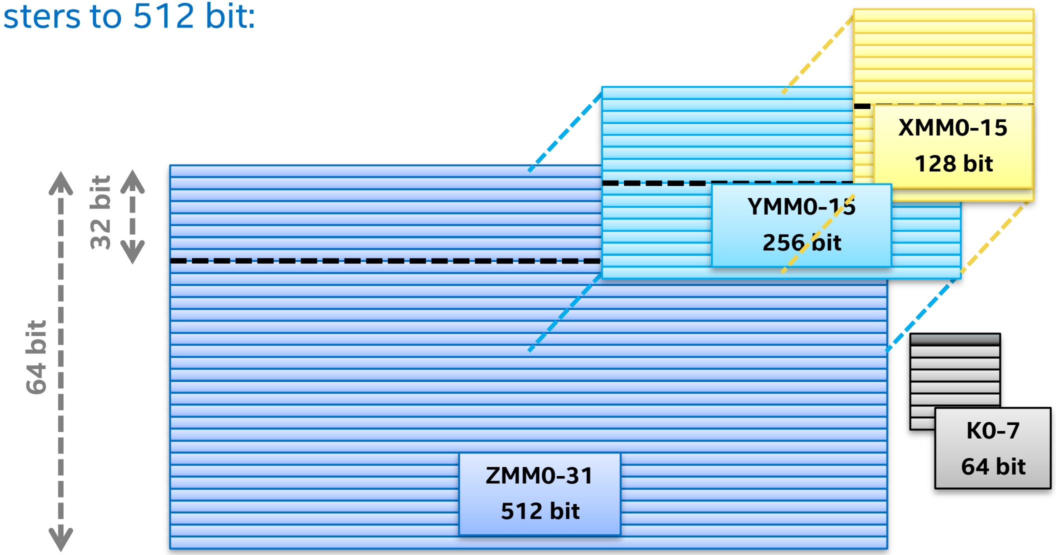


Intel® AVX and AVX-512 Registers

AVX is a 256 bit vector extension to SSE:



AVX-512 extends previous AVX and SSE registers to 512 bit:



OS support is required

Agenda

- Introduction
- Auto-vectorization of Intel Compilers
  - **Basic Vectorization Switches**
  - **Vectorization Hints**
  - **Validating Vectorization Success**
  - **Optimization Report**
- Reasons for Vectorization Failures and Inefficiency

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

Compiler:
Auto-vectorization hints (`#pragma vector`, …)

Compiler:
OpenMP* SIMD directives

SIMD intrinsic class
(e.g.: `F32vec`, `F64vec`, …)

Vector intrinsic
(e.g.: `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

Assembler code
(e.g.: `[v]addps`, `[v]addss`, …)

Ease of use

Programmer control

Auto-vectorization of Intel Compilers

```
void add(double *A, double *B, double *C)
{
  for (int i = 0; i < 1000; i++)
  C[i] = A[i] + B[i];
}
```

```
subroutine add(A, B, C)
  real*8 A(1000), B(1000), C(1000)
  do i = 1, 1000
      C(i) = A(i) + B(i)
  end do
end
```

Intel® SSE4.2

```
.B2.14:
movups    xmm1, XMMWORD PTR [edx+ebx*8]
movups    xmm3, XMMWORD PTR [16+edx+ebx*8]
movups    xmm5, XMMWORD PTR [32+edx+ebx*8]
movups    xmm7, XMMWORD PTR [48+edx+ebx*8]
movups    xmm0, XMMWORD PTR [ecx+ebx*8]
movups    xmm2, XMMWORD PTR [16+ecx+ebx*8]
movups    xmm4, XMMWORD PTR [32+ecx+ebx*8]
movups    xmm6, XMMWORD PTR [48+ecx+ebx*8]
addpd     xmm1, xmm0
addpd     xmm3, xmm2
addpd     xmm5, xmm4
addpd     xmm7, xmm6
movups    XMMWORD PTR [eax+ebx*8], xmm1
movups    XMMWORD PTR [16+eax+ebx*8], xmm3
movups    XMMWORD PTR [32+eax+ebx*8], xmm5
movups    XMMWORD PTR [48+eax+ebx*8], xmm7
add       ebx, 8
cmp       ebx, esi
jb        .B2.14
...
```

Intel® AVX

```
.B2.15
vmovupd   ymm0, YMMWORD PTR [ebx+eax*8]
vmovupd   ymm2, YMMWORD PTR [32+ebx+eax*8]
vmovupd   ymm4, YMMWORD PTR [64+ebx+eax*8]
vmovupd   ymm6, YMMWORD PTR [96+ebx+eax*8]
vaddpd    ymm1, ymm0, YMMWORD PTR [edx+eax*8]
vaddpd    ymm3, ymm2, YMMWORD PTR [32+edx+eax*8]
vaddpd    ymm5, ymm4, YMMWORD PTR [64+edx+eax*8]
vaddpd    ymm7, ymm6, YMMWORD PTR [96+edx+eax*8]
vmovupd   YMMWORD PTR [esi+eax*8], ymm1
vmovupd   YMMWORD PTR [32+esi+eax*8], ymm3
vmovupd   YMMWORD PTR [64+esi+eax*8], ymm5
vmovupd   YMMWORD PTR [96+esi+eax*8], ymm7
add       eax, 16
cmp       eax, ecx
jb        .B2.15
```

Basic Vectorization Switches I

Linux*, macOS*: **-x<code>**, Windows*: **/Qx<code>**

- Might enable Intel processor specific optimizations
- Processor-check added to "main" routine:
  Application errors in case SIMD feature missing or non-Intel processor with appropriate/informative message

<code> indicates a feature set that compiler may target (including instruction sets and optimizations)

Microarchitecture code names: BROADWELL, HASWELL, IVYBRIDGE, KNL, KNM, SANDYBRIDGE, SILVERMONT, SKYLAKE, SKYLAKE-AVX512

SIMD extensions: CORE-AVX512, CORE-AVX2, CORE-AVX-I, AVX, SSE4.2, etc.

Example:

```
icc -xCORE-AVX2 test.c
ifort -xSKYLAKE test.f90
```

Basic Vectorization Switches II

Linux*, macOS*: **-ax<code>**, Windows*: **/Qax<code>**

- Multiple code paths: baseline and optimized/processor-specific
- Optimized code paths for Intel processors defined by **<code>**
- Multiple SIMD features/paths possible, e.g.: **-axSSE2,AVX**
- Baseline code path defaults to **-msse2** (**/arch:sse2**)
- The baseline code path can be modified by **-m<code>** or **-x<code>** (**/arch:<code>** or **/Qx<code>**)
- Example: `icc` **-axCORE-AVX512 -xAVX** `test.c`

  `icc` **-axCORE-AVX2,CORE-AVX512** `test.c`

Linux*, macOS*: **-m<code>**, Windows*: **/arch:<code>**

- No check and no specific optimizations for Intel processors:
  Application optimized for both Intel and non-Intel processors for selected SIMD feature
- Missing check can cause application to fail in case extension not available

Control Vectorization I

Disable vectorization:

- Globally via switch:
  Linux*, macOS*: **`-no-vec`**, Windows*: **`/Qvec-`**
- For a single loop:
  C/C++: **`#pragma novector`**, Fortran: **`!DIR$ NOVECTOR`**
- Compiler still can use some SIMD features

Using vectorization:

- Globally via switch (default for optimization level 2 and higher):
  Linux*, macOS*: **`-vec`**, Windows*: **`/Qvec`**
- Vectorize even if compiler doesn't expect a performance benefit:
  C/C++: **`#pragma vector always`**, Fortran: **`!DIR$ VECTOR ALWAYS`**

Control Vectorization II

Verify vectorization:

- Globally:
  Linux*, macOS*: `-qopt-report`, Windows*: `/Qopt-report`

- Abort compilation if loop cannot be vectorized:
  C/C++: `#pragma vector always assert`
  Fortran: `!DIR$ VECTOR ALWAYS ASSERT`

Advanced:

- Ignore Vector DEPendencies (IVDEP):
  C/C++: `#pragma ivdep`
  Fortran: `!DIR$ IVDEP`
- "Enforce" vectorization:
  C/C++: `#pragma omp simd ...`
  Fortran: `!$OMP SIMD ...`

  Developer is responsible to verify the correctness of the code
  Enabled with option (default):
  Linux*, macOS*: `-qopenmp-simd`
  Windows*: `/Qopenmp-simd`

Validating Vectorization Success I

Optimization report:

- Linux*, macOS*: `-qopt-report=<n>`, Windows*: `/Qopt-report:<n>`
  `n`: 0, ..., 5 specifies level of detail; 2 is default (more later)
- Prints optimization report with vectorization analysis

Optimization report phase:

- Linux*, macOS*: `-qopt-report-phase=<p>`,
  Windows*: `/Qopt-report-phase:<p>`
- `<p>` is `all` by default; use `vec` for just the vectorization report

Optimization report file:

- Linux*, macOS*: `-opt-report-file=<f>`, Windows*: `/Qopt-report-file:<f>`
- `<f>` can be `stderr`, `stdout` or a file (default: *.optrpt)

Validating Vectorization Success II

Assembler code inspection (Linux*, macOS*: `-S`, Windows*: `/Fa`):

- Most reliable way and gives all details of course
- Check for scalar/packed or (E)VEX encoded instructions:
  Assembler listing contains source line numbers for easier navigation
- Compiling with `-qopt-report-embed` (Linux*, macOS*) or `/Qopt-report-embed` (Windows*) helps interpret assembly code

Intel® Advisor

Optimization Report Example

Example novec.f90:

```
1: subroutine fd(y)
2:   integer :: i
3:   real, dimension(10), intent(inout) :: y
4:   do i=2,10
5:     y(i) = y(i-1) + 1
6:   end do
7: end subroutine fd
```

```
$ ifort novec.f90 -c -qopt-report=5 -qopt-report-phase=vec
ifort: remark #10397: optimization reports are generated in *.optrpt files in the output location

$ cat novec.optrpt
…
Begin optimization report for: FD

    Report from: Vector optimizations [vec]


LOOP BEGIN at novec.f90(4,3)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed FLOW dependence between y(i) (5:5) and y(i-1) (5:5)
LOOP END
…
```

Agenda

- Introduction
- Auto-vectorization of Intel® Compilers
- Reasons for Vectorization Failures and Inefficiency
  - **Data Dependence**
  - **Alignment**
  - **Unsupported Loop Structure**
  - **Non-Unit Stride Access**
  - **Mathematical Functions**

Reasons for Vectorization Failures and Inefficiency

Most frequent reasons:

Data dependence

Alignment

Unsupported loop structure

Non-unit stride access

Function calls

Non-vectorizable mathematical functions

All those are common and will be explained in detail next!

Data Dependency and vectorization

Flow Dependency	Anti Dependency	Output Dependency
`X = …` `… = X`	`… = X` `X = …`	`X = …` `X = …`
read-after-write RAW	write-after-read WAR	write-after-write WAW

```
DO I = 1, 10000
  A(I) = B(I) * 17

  X(I+1) = X(I) + A(I)
ENDDO
```

Loop-independent dependence

Loop-carried dependence

Example:
Despite cyclic dependency, the loop can be vectorized for SSE or AVX in case of VL being max. 3 times the data type size of array **A**.

```
DO I = 1, N
  A(I + 3) = A(I) + C
END DO
```

Failing Disambiguation

Many potential dependencies detected by the compiler result from unresolved memory disambiguation:

The compiler has to be conservative and has to assume the worst case regarding "aliasing"!

Example:

```
void scale(int *a, int *b)
{
    for (int i = 0; i < 10000; i++) b[i] = z * a[i];
}
```

Without additional information (like inter-procedural knowledge) the compiler has to assume a and b to be aliased!

Use directives, switches and attributes to aid disambiguation!

This is programming language and operating system specific

Use with care as the compiler might generate incorrect code in case the hints are not fulfilled!

Disambiguation Hints I

Disambiguating memory locations of pointers in C99:
Linux*, macOS*: `-std=c99`, Windows*: `/Qstd=c99`

Intel® C++ Compiler also allows this for other modes
(e.g. `-std=c89`, `-std=c++0x`, …), too - **not standardized**, though:
Linux*, macOS*: `-restrict`, Windows*: `/Qrestrict`

Declaring pointers with keyword `restrict` asserts compiler that they only reference individually assigned, non-overlapping memory areas

Also true for any result of pointer arithmetic (e.g. `ptr + 1` or `ptr[1]`)

Examples:

```
void scale(int *a, int *restrict b)
{
    for (int i = 0; i < 10000; i++) b[i] = z * a[i];
}

void mult(int a[][NUM], int b[restrict][NUM])
{ ... }
```

Disambiguation Hints II

Directive:

`#pragma ivdep` (C/C++) or **`!DIR$ IVDEP`** (Fortran)

For C/C++:

- Assume no aliasing at all (dangerous!):
  Linux*, macOS*: **`-fno-alias`**, Windows*: **`/Oa`**
- Assume ISO C Standard aliasing rules:
  Linux*, macOS*: **`-ansi-alias`**, Windows*: **`/Qansi-alias`**
  Default on Linux, not on Windows
  - Turns on ANSI aliasing checker
- No aliasing between function arguments:
  Linux*, macOS*: **`-fargument-noalias`**, Windows*: **`/Qalias-args-`**
- No aliasing between function arguments and global storage:
  Linux*, macOS*: **`-fargument-noalias-global`**, Windows*: N/A

Multiversioning for data dependence

Example `test.cpp`:

```
void add(double *A, double *B, double *C)
{
   for (int i = 0; i < 1000; i++)
         C[i] = A[i] + B[i];
}
```

```
$ icpc test.cpp -c -qopt-report=5 -qopt-report-phase=vec
icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location
$ cat test.optrpt
...
```

Multiversioning for data dependence

```
LOOP BEGIN at  test.cpp(3,2)
Multiversioned v1
test.cpp(4,3):remark #15388: vectorization support: reference C[i] has aligned access
test.cpp(4,3):remark #15389: vectorization support: reference A[i] has unaligned access
test.cpp(4,3):remark #15388: vectorization support: reference B[i] has aligned access
test.cpp(3,2):remark #15381: vectorization support: unaligned access used inside loop body
test.cpp(3,2):remark #15305: vectorization support: vector length 2
test.cpp(3,2):remark #15399: vectorization support: unroll factor set to 4
test.cpp(3,2):remark #15309: vectorization support: normalized vectorization overhead 0.607
test.cpp(3,2):remark #15300: LOOP WAS VECTORIZED
test.cpp(3,2):remark #15442: entire loop may be executed in remainder
test.cpp(3,2):remark #15448: unmasked aligned unit stride loads: 1
test.cpp(3,2):remark #15449: unmasked aligned unit stride stores: 1
test.cpp(3,2):remark #15450: unmasked unaligned unit stride loads: 1
test.cpp(3,2):remark #15475: --- begin vector cost summary ---
test.cpp(3,2):remark #15476: scalar cost: 8
test.cpp(3,2):remark #15477: vector cost: 3.500
test.cpp(3,2):remark #15478: estimated potential speedup: 2.250
test.cpp(3,2):remark #15488: --- end vector cost summary ---
LOOP END
...
LOOP BEGIN at test.cpp(3,2)
Multiversioned v2
test.cpp(3,2):remark #15304: loop was not vectorized: non-vectorizable loop instance from
multiversioning
LOOP END
```

Optimization Report – An Example

$ icc -c -xcommon-avx512 **-qopt-report=3** -qopt-report-phase=loop,vec foo.c

Creates foo.optrpt summarizing which optimizations the compiler performed or tried to perform.
Level of detail from 0 (no report) to 5 (maximum).
-qopt-report-phase=loop,vec asks for a report on vectorization and loop optimizations only
Extracts:

```
LOOP BEGIN at foo.c(4,3)
Multiversioned v1
  remark #25228: Loop multiversioned for Data Dependence…
  remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 1
  remark #15451: unmasked unaligned unit stride stores: 1
  ….   (loop cost summary)  ….
LOOP END

LOOP BEGIN at foo.c(4,3)
<Multiversioned v2>
  remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning
LOOP END
```

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 512; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

Optimization Report – An Example

```
$ icc -c -xcommon-avx512  -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias foo.c
…
LOOP BEGIN at foo.c(4,3)
…
   remark #15417: vectorization support: number of FP up converts: single precision to double precision 1   [ foo.c(5,17) ]
   remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   [ foo.c(5,8) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 111
   remark #15477: vector cost: 10.310
   remark #15478: estimated potential speedup: 10.740
   remark #15482: vectorized math library calls: 1
   remark #15487: type converts: 2
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=32
LOOP END
```

report to stderr
instead of foo.optrpt

```
#include <math.h>
void foo (float * theta, float * sth)   {
  int i;
  for (i = 0; i < 512; i++)
      sth[i] = sin(theta[i]+3.1415927);
}
```

https://godbolt.org/z/aMtp9T

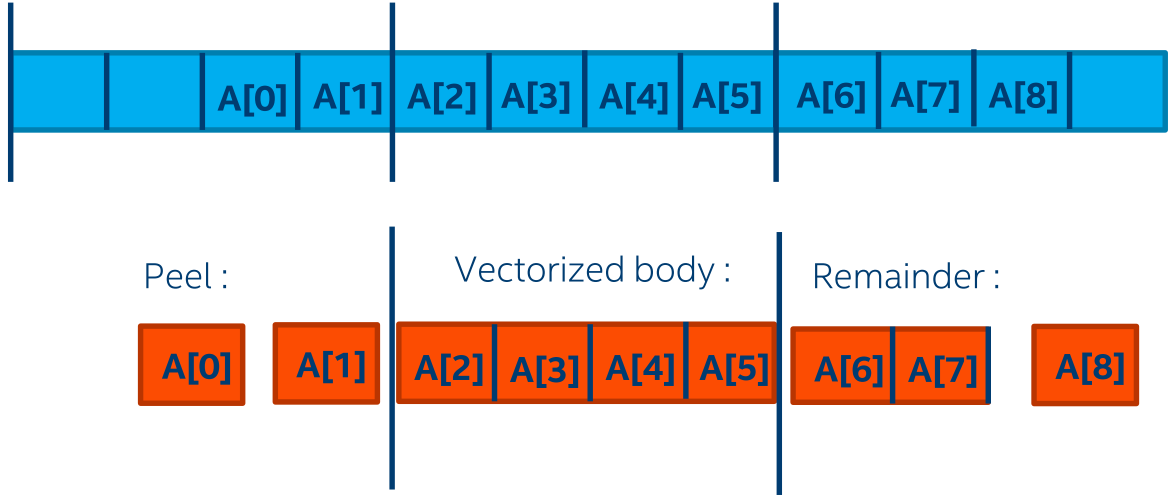
Optimization Report – An Example

```
$ icc -S -xcommon-avx512 -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias foo.c
LOOP BEGIN at foo2.c(4,3)
...
remark #15305: vectorization support: vector length 32
remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 1
  remark #15451: unmasked unaligned unit stride stores: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 109
  remark #15477: vector cost: 5.250
  remark #15478: estimated potential speedup: 20.700
  remark #15482: vectorized math library calls: 1
  remark #15488: --- end vector cost summary ---
  remark #25015: Estimate of max trip count of loop=32
LOOP END

$ grep sin foo.s
    call    __svml_sinf16_b3
    call    __svml_sinf16_b3
```

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 512; i++)
    sth[i] =
sinf(theta[i]+3.1415927f);
}
```

Compiler helps with alignment

SSE: 16 bytes
AVX: 32 bytes
AVX512 64 bytes



Compiler can split loop in 3 parts to have aligned access in the loop body

Data Alignment for C/C++

Aligned heap memory allocation by intrinsic/library call:

`void* aligned_alloc( std::size_t alignment, std::size_t size );` (since C++17)

`void* _mm_malloc(int size, int base)`

Linux*, macOS* only:

`int posix_memaligned(void **p, size_t base, size_t size)`

Automatically allocate memory with the alignment of that type using new operator:

`#include <aligned_new>`

Align attribute for variable declarations:

`alignas` specifier (since C++11):

alignas(64) char line[128];

Linux*, macOS*, Windows*: `__declspec(align(base)) <var>`

Linux*, macOS*: `<var> __attribute__((aligned(base)))`

Portability caveat:

`__declspec` is not known for GCC and `__attribute__` not for Microsoft Visual Studio*!

Compiler Alignment Hints for C/C++

Hint that start address of an array is aligned (Intel Compiler only):

```
__assume_aligned(<array>, base)
```

```
#pragma vector [aligned|unaligned]
```

- Only for Intel Compiler
- Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
- **Use with care:**
  The assertion must be satisfied for all(!) data accesses in the loop!

Problems Defining Alignment

```
void matvec(double a[][ROWWIDTH], double b[], double c[])
{
  int i, j;
  for(i = 0; i < size1; i++) {
    b[i] = 0;
#pragma vector aligned
    for(j = 0; j < size2; j++)
      b[i] += a[i][j] * c[j];
  }
}
```

- Let's assume **a**, **b** and **c** are be declared 16 byte aligned in calling routine
- **Question:** Would this be correct when compiled for Intel® SSE2?
- **Answer:** It depends on **ROWWIDTH**!
  - **ROWWIDTH** is even: Yes
  - **ROWWIDTH** is odd: No, vectorized code fails with alignment error after first row!
- **Solution:**
  Instead of pragma, use `__assume_aligned(<array>, base)`. This refers to the start address only.

Hands-on exercises

git clone https://github.com/ivorobts/compiler-optimization.git

Use Vectorization_Lab.pdf for instructions

- C++/Fortran – choose what you prefer

Unsupported Loop Structure

https://godbolt.org/z/TAXULn

Loops where compiler does not know the iteration count:

- Upper/lower bound of a loop are not loop-invariant
- Loop stride is not constant
- Early bail-out during iterations (e.g. **break**, exceptions, etc.)
- Too complex loop body conditions for which no SIMD feature instruction exists
- Loop dependent parameters are globally modifiable during iteration (language standards require load and test for each iteration)

Transform is possible, e.g.:

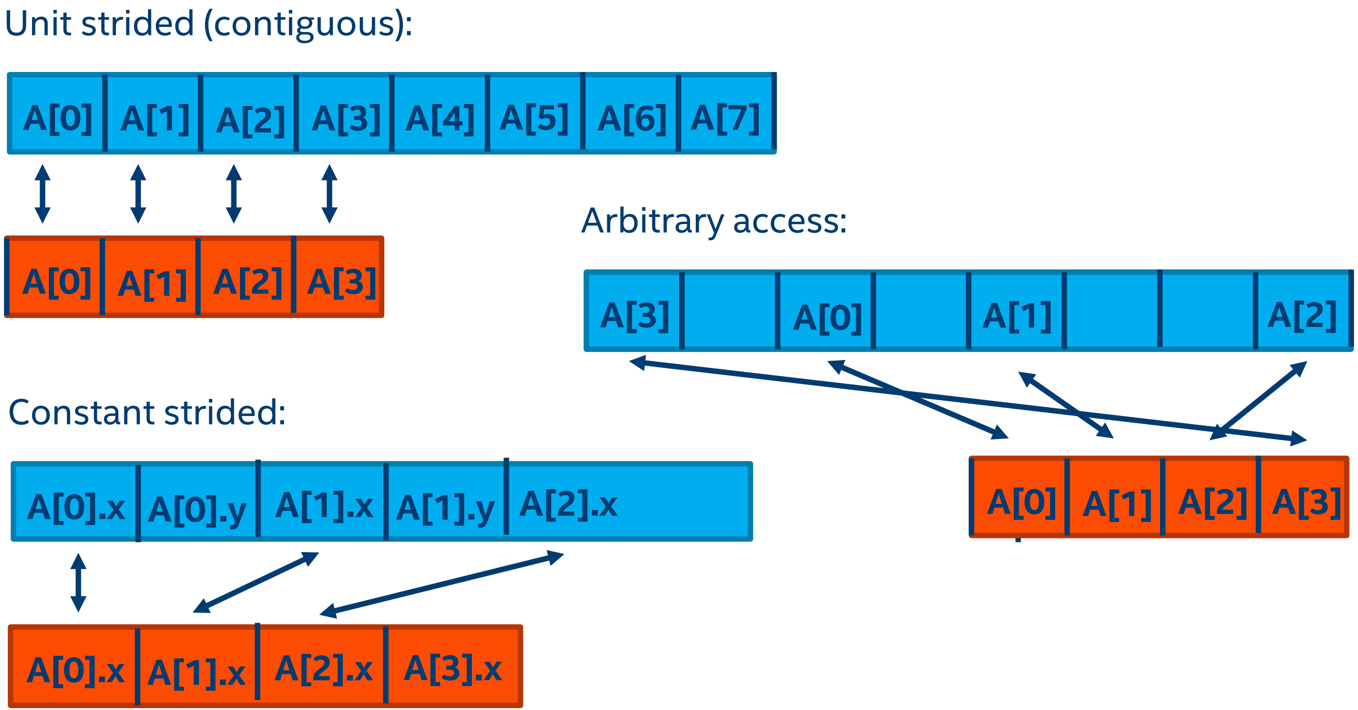
```
struct _x { int d; int bound; };

void doit(int *a, struct _x *x)
{

  for(int i = 0; i < x->bound; i++)
    a[i] = 0;
}
```

→

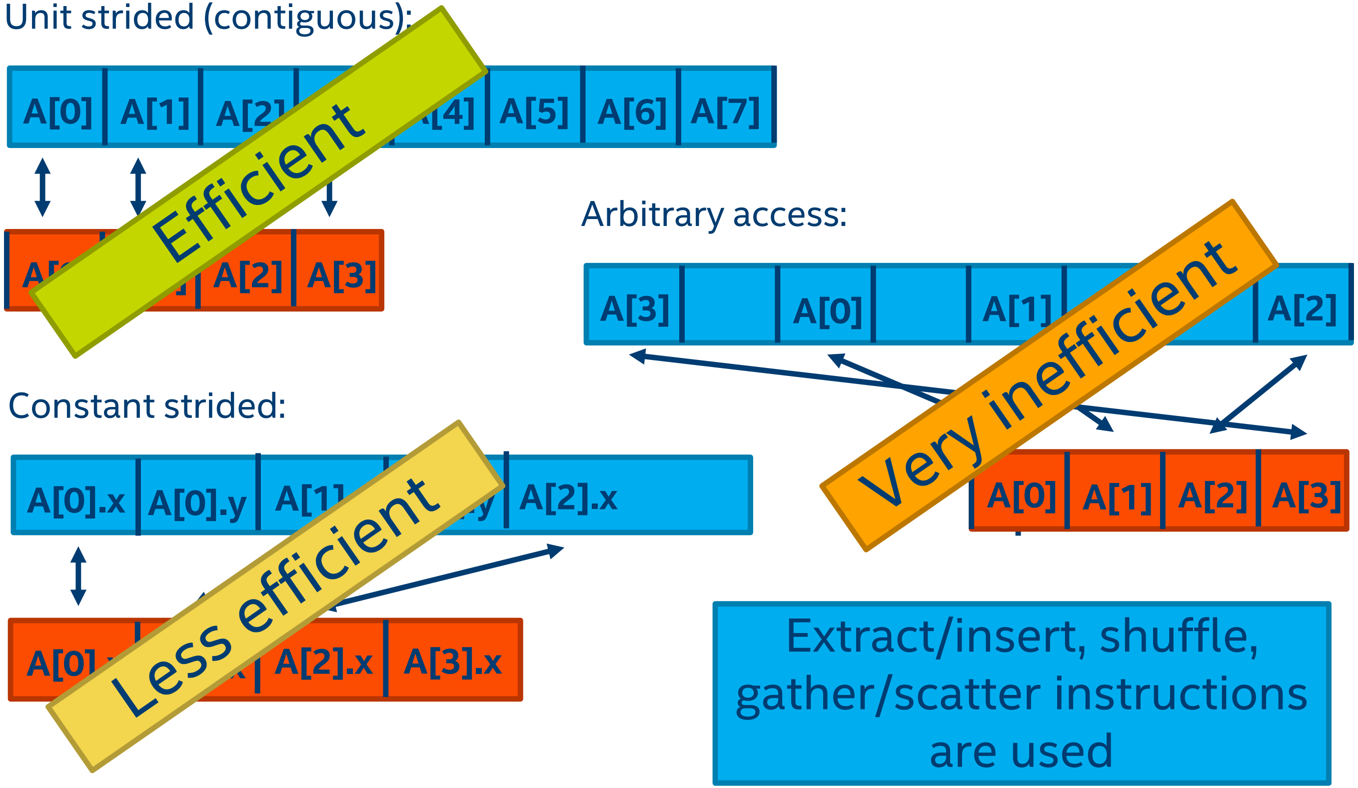
```
struct _x { int d; int bound; };

void doit(int *a, struct _x *x)
{
  int local_ub = x->bound;
  for(int i = 0; i < local_ub; i++)
    a[i] = 0;
}
```

```
loop was not vectorized: loop control variable i was found, but loop iteration
count cannot be computed before executing the loop
```

Memory access patterns



Memory access patterns

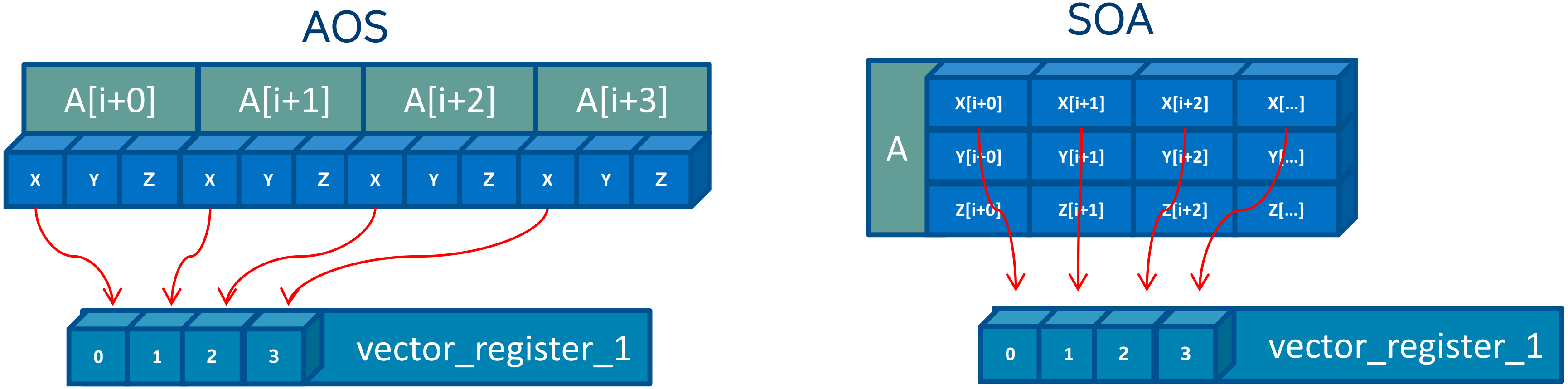


Extract/insert, shuffle, gather/scatter instructions are used

What is Intel® SDLT?

The SIMD Data Layout Template library is a C++11 template library to quick convert *Array of Structures* to *Structure of Arrays* representation

SDLT vectorizes your code by making memory access contiguous, which can lead to more efficient code and better performance



https://tinyurl.com/intelsdlt

Use function calls inside loop

Success of in-lining can be verified using the optimization report:
Linux*, macOS*: `-qopt-report=<n> -qopt-report-phase=ipo`
Windows*: `/Qopt-report:<n> /Qopt-report-phase:ipo`

Intel compilers offer a large set of switches, directives and language extensions to control in-lining globally or locally, e.g.:

- `#pragma [no]inline` (C/C++), `!DIR$ [NO]INLINE` (Fortran):
  Instructs compiler that all calls in the following statement can be in-lined or may never be in-lined
- `#pragma forceinline` (C/C++), `!DIR$ FORCEINLINE` (Fortran):
  Instructs compiler to ignore the heuristic for in-lining and to inline all calls in the following statement
- See section “Inlining Options” in compiler manual for full list of options

IPO offers additional advantages to vectorization

Vectorizable Mathematical Functions

Calls to most mathematical functions in a loop body can be vectorized using "Short Vector Math Library" (SVML):

- SVML (**libsvml**) provides vectorized implementations of different mathematical functions
- Optimized for latency compared to the VML library component of Intel® MKL which realizes same functionality but optimized for throughput

Routines in **libsvml** can also be called explicitly, using intrinsics (C/C++)

These mathematical functions are currently supported:

acos	acosh	asin	asinh	atan	atan2	atanh	cbrt
ceil	cos	cosh	erf	erfc	erfinv	exp	exp2
fabs	floor	fmax	fmin	log	log10	log2	pow
round	sin	sinh	sqrt	tan	tanh	trunc	

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

Compiler:
Auto-vectorization hints (`#pragma vector`, …)

Compiler:
OpenMP* SIMD directives

SIMD intrinsic class
(e.g.: `F32vec`, `F64vec`, …)

Vector intrinsic
(e.g.: `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

Assembler code
(e.g.: `[v]addps`, `[v]addss`, …)

Ease of use ↕ **Programmer control**

Obstacles to Auto-Vectorization

Multiple loop exits

- Or trip count unknown at loop entry

Dependencies between loop iterations

- Mostly, avoid read-after-write “flow” dependencies

Function or subroutine calls

- Except where inlined

Nested (Outer) loops

- Unless inner loop fully unrolled

Complexity

- Too many branches
- Too hard or time-consuming for compiler to analyze

https://software.intel.com/articles/requirements-for-vectorizable-loops

OpenMP* SIMD Programming

Vectorization is so important
→ consider explicit vector programming

Modeled on OpenMP* for threading (explicit parallel programming)

Enables reliable vectorization of complex loops the compiler can't auto-vectorize
- E.g. outer loops

Directives are commands to the compiler, not hints
- E.g. #pragma omp simd or !$OMP SIMD
- Compiler does no dependency and cost-benefit analysis !!
- **Programmer is responsible for correctness** (like OpenMP threading)
  - E.g. PRIVATE, REDUCTION or ORDERED clauses

Incorporated in OpenMP since version 4.0 ⇒ portable
- -qopenmp or -qopenmp-simd (default starting 19.0 version) to enable

OpenMP* SIMD pragma

https://godbolt.org/z/qfpHxb

Use #pragma omp simd with -qopenmp-simd

```
void addit(double* a, double* b, int
m, int n, int x)
{
  for (int i = m; i < m+n; i++)   {
        a[i] = b[i] + a[i-x];
  }
}
```

loop was not vectorized:
existence of vector dependence.

```
void addit(double* a, double * b, int m,
int n, int x)
{
#pragma omp simd   // I know x<0
  for (int i = m; i < m+n; i++)   {
        a[i] = b[i] + a[i-x];
  }
}
```

SIMD LOOP WAS VECTORIZED

Use when you **KNOW** that a given loop is safe to vectorize

The Intel® Compiler will vectorize if at all possible

- (ignoring dependency or efficiency concerns)
- Minimizes source code changes needed to enforce vectorization

Clauses for OMP SIMD directives

The programmer (i.e. you!) is responsible for correctness

- Just like for race conditions in loops with OpenMP* threading

Available clauses:

- PRIVATE
- LASTPRIVATE
- REDUCTION

} like OpenMP for threading

- COLLAPSE (for nested loops)
- LINEAR (additional induction variables)
- SIMDLEN (preferred number of iterations to execute concurrently)
- SAFELEN (max iterations that can be executed concurrently)
- ALIGNED (tells compiler about data alignment)

Example: Outer Loop Vectorization

```
#ifdef  KNOWN_TRIP_COUNT
#define MYDIM 3
#else                                // pt     input  vector of points
#define MYDIM nd                     // ptref  input  reference point
#endif                               // dis    output vector of distances
#include <math.h>

void dist( int n, int nd, float pt[][MYDIM], float dis[], float ptref[]) {
/* calculate distance from data points to reference point */

#pragma omp simd
      for (int ipt=0; ipt<n; ipt++) {
            float d = 0.;

            for (int j=0; j<MYDIM; j++) {
                  float t = pt[ipt][j] - ptref[j];
                  d+= t*t;
            }

            dis[ipt] = sqrtf(d);
      }
}
```

Outer loop with high trip count

Inner loop with low trip count

https://godbolt.org/z/nfS2c6

Outer Loop Vectorization

```
icc -std=c99 -xavx -qopt-report-phase=loop,vec -qopt-report-file=stderr -c dist.c
…
LOOP BEGIN at dist.c(26,2)
   remark #15542: loop was not vectorized: inner loop was already vectorized
…
   LOOP BEGIN at dist.c(29,3)
      remark #15300: LOOP WAS VECTORIZED
```

We can vectorize the outer loop by activating the pragma using -qopenmp-simd

```
#pragma omp simd
```

Would need private clause for d and t if declared outside SIMD scope

```
icc -std=c99 -xavx -qopenmp-simd -qopt-report-phase=loop,vec -qopt-report-file=stderr -qopt-report=4 -c dist.c
…
LOOP BEGIN at dist.c(26,2)
   remark #15328: … non-unit strided load was emulated for the variable <pt[ipt][j]>, stride is unknown to compiler
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   LOOP BEGIN at dist.c(29,3)
      remark #25460: No loop optimizations reported
```

Unrolling the Inner Loop

There is still an inner loop.

If the trip count is fixed and the compiler knows it, the inner loop can be fully unrolled.
Outer loop vectorization is more efficient also because stride is now known

```
icc -std=c99 -xavx -qopenmp-simd -DKNOWN_TRIP_COUNT -qopt-report-phase=loop,vec -
qopt-report-file=stderr –qopt-report=4 -c dist.c
…
LOOP BEGIN at dist.c(26,2)
  remark #15328: vectorization support: non-unit strided load was emulated for the variable <pt[ipt][j]>,
                          stride is 3   [ dist.c(30,14) ]
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

  LOOP BEGIN at dist.c(29,3)
    remark #25436: completely unrolled by 3   (pre-vector)
  LOOP END
LOOP END
```

Loops Containing Function Calls

Function calls can have side effects that introduce a loop-carried dependency, preventing vectorization

Possible remedies:

- Inlining
  - best for small functions
  - Must be in same source file, or else use -ipo
- OMP SIMD pragma or directive to vectorize rest of loop, while preserving scalar calls to function (last resort)
- SIMD-enabled functions
  - Good for large, complex functions and in contexts where inlining is difficult
  - Call from regular "for" or "DO" loop
  - In Fortran, adding "ELEMENTAL" keyword allows SIMD-enabled function to be called with array section argument

SIMD-enabled Function

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

y, z, xp, yp and zp are constant, x can be a vector

```
#pragma omp declare simd uniform(y,z,xp,yp,zp)
float func(float x, float y, float z, float xp, float yp, float zp)
{
float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp) + (z-zp)*(z-zp);
  denom = 1./sqrtf(denom);
  return denom;
}
…
#pragma omp simd  private(x)  reduction(+:sumx)
for (i=1; i<nx; i++) {
    x = x0 + (float) i * h;
    sumx = sumx + func(x, y, z, xp, yp, zp);
  }
```

FUNCTION WAS VECTORIZED with …

These clauses are required for correctness, just like for OpenMP*

SIMD LOOP WAS VECTORIZED.

#pragma omp simd may not be needed in simpler cases

Special Idioms

Dependency on an earlier iteration usually makes vectorization unsafe

- Some special patterns can still be handled by the compiler
  - Provided the compiler recognizes them (auto-vectorization)
    - Often works only for simple, ‘clean’ examples
  - Or the programmer tells the compiler (explicit vector programming)
    - May work for more complex cases
  - Examples: reduction, compress/expand, search, histogram/scatter, minloc
- Sometimes, the main speed-up comes from vectorizing the rest of a large loop, more than from vectorization of the idiom itself

Reduction – simple example

Auto-vectorizes with any instruction set:

icc -std=c99 -O2 -qopt-report-phase=loop,vec -qopt-report-file=stderr reduce.c;

…

LOOP BEGIN at reduce.c(17,6))

remark #15300: LOOP WAS VECTORIZED

```
double reduce(double a[], int na) {
/*    sum all positive elements of a   */
      double sum = 0.;
      for (int ia=0; ia <na; ia++)   {
          if (a[ia] > 0.)   sum += a[ia];    // sum causes cross-iteration dependency
      }
      return sum;
}
```

Reduction – when auto-vectorization doesn't work

icc -std=c99 -O2 -fp-model precise -qopt-report-phase=loop,vec -qopt-report-file=stderr reduce.c;
…
LOOP BEGIN at reduce.c(17,6))
remark #15331: loop was not vectorized: precise FP model implied by the command line or a directive prevents vectorization. Consider using fast FP model [reduce.c(18,26)

Vectorization would change order of operations, and hence the result

- Can use a SIMD pragma to override and vectorize:

```
#pragma omp simd reduction(+:sum)
     for (int ia=0; ia <na; ia++)
{
          sum += …
```

Without the reduction clause, results would be incorrect because of the flow dependency. See "SIMD-Enabled Function" section for another example.

icc -std=c99 -O2 -fp-model precise –qopenmp-simd -qopt-report-file=stderr reduce.c;
LOOP BEGIN at reduce.c(18,6)
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

Exercise – nbody-demo/ver4

- Go to the folder nbody-demo/ver4
- Type *make* to compile code.
- Type *make run* to run the test and measure the timing.
- Please have a look at the compiler report.

Compiler report result

```
LOOP BEGIN at GSimulation.cpp(145,6)
         remark #15389: vectorization support: reference this->particles->pos_x[j] has
unaligned access   [ GSimulation.cpp(151,8) ]
…
         remark #15389: vectorization support: reference this->particles->mass[j] has
unaligned access   [ GSimulation.cpp(160,20) ]
         remark #15381: vectorization support: unaligned access used inside loop body
         remark #15305: vectorization support: vector length 8
         remark #15309: vectorization support: normalized vectorization overhead 1.055
         remark #15300: LOOP WAS VECTORIZED
         remark #15321: Compiler has chosen to target XMM/YMM vector. Try using -qopt-
zmm-usage=high to override
         remark #15450: unmasked unaligned unit stride loads: 4
         remark #15475: --- begin vector cost summary ---
         remark #15476: scalar cost: 118
         remark #15477: vector cost: 13.620
         remark #15478: estimated potential speedup: 7.910
         remark #15488: --- end vector cost summary ---
      LOOP END
```

Exercise – nbody-demo/ver5

- Go to the folder *nbody-demo/ver5*
- Open the file *GSimulation.cpp*: all the memory allocations have been replaced with *_mm_malloc*
- Type *make* to compile code.
- Type *make run* to run the test and measure the timing.
- Please have a look at the compiler report.
- Try to recompile the code removing the option –DASSUME_ALIGN from the *Makefile*. Can you explain what is going on?

Exercise – nbody-demo/ver5

- Play with the alignment:
  - alignment 64 bytes also with 32. 16 does not align
- Modify into the *Makefile* the OPTFLAG (ISA optimization) to *–xCORE-AVX2*
  - What is changing in the alignment?
  - What is the performance?
- Type *make run* to run the test and measure the timing.

Exercise – nbody-demo/ver5

- Recompile the code enabling the usage of the zmm registers
  - *make ZMM=yes*
- Can you explain why the performance is higher?
  - Please have a look at the compiler report for hints.

Exercise – nbody-demo/ver5

- Recompile the code enabling the usage of the zmm registers
  - *make ZMM=yes*
- Can you explain why the performance is higher?
  - Please have a look at the compiler report for hints.

- *With the usage of AVX512 and ZMM registers, we have 22% more performance!*

Legal Disclaimers and Optimization Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

All products, platforms, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

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. § For more information go to www.intel.com/benchmarks.

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.

The Intel Core and Itanium processor families may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

The benchmark results reported may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations.

The code names Arrandale, Bloomfield, Boazman, Boulder Creek, Calpella, Chief River, Clarkdale, Cliffside, Cougar Point, Gulftown, Huron River, Ivy Bridge, Kilmer Peak, King's Creek, Lewisville, Lynnfield, Maho Bay, Montevina, Montevina Plus, Nehalem, Penryn, Puma Peak, Rainbow Peak, Sandy Bridge, Sugar Bay, Tylersburg, and Westmere presented in this document are only for use by Intel to identify a product, technology, or service in development, that has not been made commercially available to the public, i.e., announced, launched or shipped. It is not a "commercial" name for products or services and is not intended to function as a trademark.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel, Intel Core, Core Inside, Itanium, and the Intel Logo are trademarks of Intel Corporation in the U.S. and other countries.

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

Intel collects and uses personal information from employees as part of SES, including capturing audio recording of sessions (both presenters and audience QA) as well as photographs and video recording of various event activities during the event. By registering and attending the SES conference, you give your consent for this capture. This includes both speakers and attendees. Intel will not retain your personal information longer than is necessary for the purposes for which it is collected.

Legal Disclaimer & Optimization Notice

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. For more complete information visit www.intel.com/benchmarks.

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

intel®

Software