Vectorization with Intel® Compilers and OpenMP* 4.0 SIMD

Dr. Mikko Byckling
Senior Application Engineer

Acknowledgements: Martyn Corden, Intel; Steve "Dr. Fortran" Lionel, ex-Intel

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

Notices and Disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

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

Intel processors of the same SKU may vary in frequency or power as a result of natural variability in the production process.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

The benchmark results reported above 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.

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.

No computer system can be absolutely secure.

Intel® Advanced Vector Extensions (Intel® AVX)* provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel® Hyper-Threading Technology available on select Intel® processors. Requires an Intel® HT Technology-enabled system. Your performance varies depending on the specific hardware and software you use. Learn more by visiting http://www.intel.com/info/hyperthreading.

© 2018 Intel Corporation. Intel, the Intel logo, Xeon and Xeon logos are trademarks of Intel Corporation in the U.S. and/or other countries.

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

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

Contents

- Vectorization with Fortran array syntax
- OpenMP* SIMD
  - OpenMP* `SIMD` construct
  - OpenMP* `DECLARE SIMD` construct
- SIMD programming patterns
  - Reduction, outer loop vectorization, compress, search and histogram loops
- Summary

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

Contents

- Vectorization with Fortran array syntax
- OpenMP* SIMD
  - OpenMP* `SIMD` construct
  - OpenMP* `DECLARE SIMD` construct
- SIMD programming patterns
  - Reduction, outer loop vectorization, compress, search and histogram loops
- Summary

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

Fortran array syntax

- The compiler will attempt to vectorize and use SIMD instructions for Fortran array syntax operations

```
INTEGER :: n, ind(n)
REAL :: a(n), b(n), c(n),d(2*n)
! Access by the whole array
a=b+c
! Access by array section
a(1:n)=a(1:n)*d(n+1:2*n)
a(1:n:2)=a(1:n:2)/b(1:n:2)
! Access by array indices
b(1:n)=2*b(ind(1:n))
```

```
remark #15300: LOOP WAS VECTORIZED
remark #15450: unmasked unaligned unit stride loads: 2
remark #15451: unmasked unaligned unit stride stores: 1
```

```
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 1
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 1
```

```
remark #15300: LOOP WAS VECTORIZED
remark #15452: unmasked strided loads: 2
remark #15453: unmasked strided stores: 1
```

```
 -> EXTERN: (10,3) _alloca
…
remark #15300: LOOP WAS VECTORIZED
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 1
remark #15462: unmasked indexed (or gather) loads: 1
```


Copyright © 2018 Intel Corporation. All rights reserved.

WHERE statement/construct

- Perform array operations only on the selected items

```
[name:] WHERE (mask-expr)
[ where-body …]
[ELSE WHERE (mask-expr) [name]
[ where-body …]
[ELSE WHERE [name] ]
[ where-body …]
END WHERE [name]
```

  where ***mask-expr*** is a logical array expression and ***where-body*** is either an *array assignment* or a nested **WHERE** statement/construct

- Masked array assignment - an array operation is performed only for elements where ***mask-expr*** is **.TRUE.**
- *Array assignment* can be performed with an elemental function

Copyright © 2018 Intel Corporation. All rights reserved.

WHERE construct: example

```
SUBROUTINE divide(n, arr1, arr2)
  INTEGER, INTENT(IN) :: n
  REAL :: arr1(n), arr2(n)
  WHERE (arr2 /= 0)
    arr1 = arr1/arr2
  ELSEWHERE
    arr1 = 0.0
  END WHERE
END SUBROUTINE divide
```

```
-> EXTERN: (4,3) _alloca
…
LOOP BEGIN at divide.F90(4,3)
…
remark #15300: LOOP WAS VECTORIZED
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 1
```

```
LOOP BEGIN at divide.F90(5,6)
…
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 1
remark #15456: masked unaligned unit stride loads: 2
remark #15457: masked unaligned unit stride stores: 1
```

```
LOOP BEGIN at divide.F90(7,6)
…
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 1
remark #15457: masked unaligned unit stride stores: 1
```


Copyright © 2018 Intel Corporation. All rights reserved.

DO CONCURRENT construct

- Execute loop (in parallel) with no restrictions on the loop execution order

  [name:] DO [,] CONCURRENT ***concur-header***
  block
  END DO [name]

  where ***concur-header*** is

  ([type-spec ::] ***concur-control-list*** **[, scalar-mask-expr])**

  and ***concur-control*** is

  index-name = init-value : final-value [: step-value]

- Additions restrictions on ***block***:
  - May not branch out of or branch into the construct. May not contain an **EXIT** statement that exits the construct, nor a **CYCLE** statement for an outer loop of the construct.
  - **RETURN** or an image control statement not allowed in the construct
  - No references to impure procedures or **IEEE_EXCEPTION** procedures

Copyright © 2018 Intel Corporation. All rights reserved.

DO CONCURRENT construct: example

```
SUBROUTINE arraysum(n, arr1, arr2)
  INTEGER, INTENT(IN) :: n
  REAL :: arr1(n,n,n), arr2(n,n,n)
  INTEGER :: i, j, k
  DO CONCURRENT (k=1:n,j=1:n,i=1:n)
     arr1(i,j,k) = arr1(i,j,k) + arr2(i,j,k)
  END DO
END SUBROUTINE arraysum
```

```
LOOP BEGIN at arraysum.F90(5,3)
…
remark #17109: LOOP WAS AUTO-PARALLELIZED
…
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 1
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 1
```


Copyright © 2018 Intel Corporation. All rights reserved.

Fortran array syntax and performance

- Factors potentially limiting performance with array syntax
  - The compiler may not be able to share register loads between subsequent array operations
  - Large arrays may overflow from cache
  - For correct semantics, the compiler may have to create array temporaries
- NOTE: With Intel® Fortran compiler, **`DO CONCURRENT`** requires an additional compiler flags to enable autoparallelization (**`-parallel`**)

Copyright © 2018 Intel Corporation. All rights reserved.

Contents

- Vectorization with Fortran array syntax
- OpenMP* SIMD
  - OpenMP* `SIMD` construct
  - OpenMP* `DECLARE SIMD` construct
- SIMD programming patterns
  - Reduction, outer loop vectorization, compress, search and histogram loops
- Summary

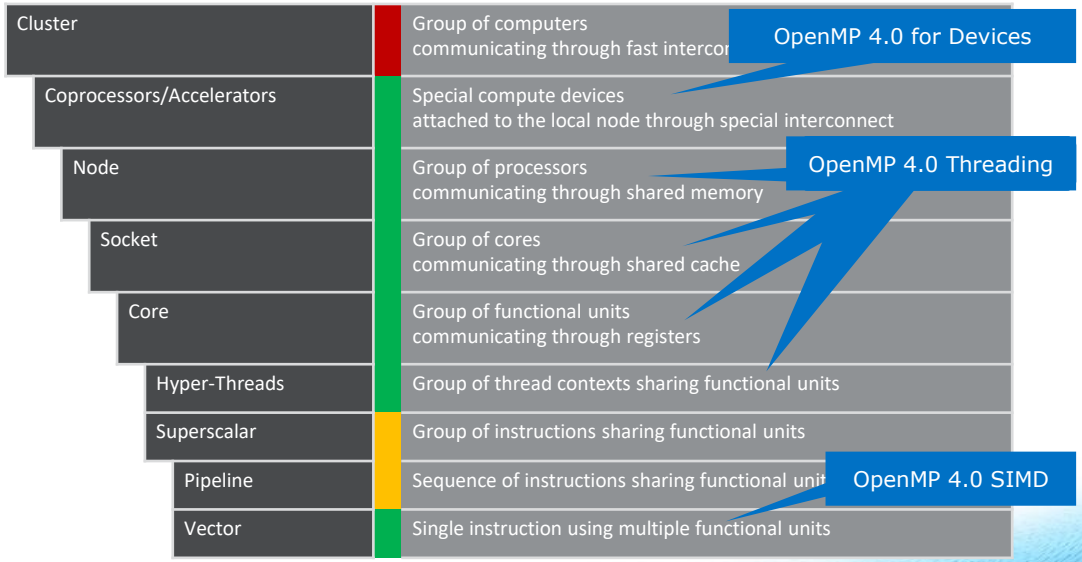
Copyright © 2018 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

OpenMP* API

- De-facto standard, OpenMP* 4.0 out since July 2013
- API for C/C++ and Fortran for shared-memory parallel programming
- Based on directives
- Portable across vendors and platforms
- Supports various types of parallelism

Copyright © 2018 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

Levels of parallelism in OpenMP 4.0



SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

Explicit vectorization

- Compiler Responsibilities
  - Allow programmer to declare that code **can** and **should** be run in SIMD
  - Generate the code the programmer asked for
- Programmer Responsibilities
  - Correctness (e.g., no dependencies, no invalid memory accesses)
  - Efficiency (e.g., alignment, loop order, masking)
- Enables efficient vectorization of **complex loops** the compiler is unable to auto-vectorize
  - Assumed vector dependencies, vectorization of outer loops, loops with function calls, etc.

Copyright © 2018 Intel Corporation. All rights reserved.

Before OpenMP 4.0 SIMD

- Programmers had to rely on auto-vectorization…
- … or to use vendor-specific extensions such as compiler pragmas (`!DIR$ vector`) etc.

```
!$omp parallel do
!DIR$ vector always
!DIR$ ivdep
DO i=1,N
  a(i) = b(i) + ...
END DO
```

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

- OpenMP 4.0 SIMD is **portable** across different compilers

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `SIMD` construct

- Vector *parallelism* is decribed with `simd` construct
  - 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
```

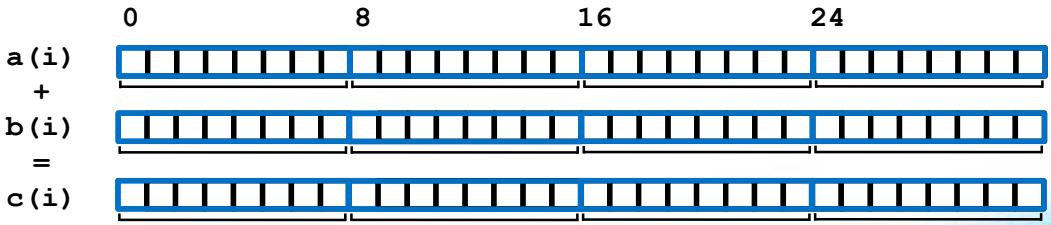
- Syntax (Fortran)

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


Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP SIMD: example

```
SUBROUTINE ssum(n, a, b, c)
  INTEGER :: n, i
  REAL(KIND=real64), POINTER, CONTIGUOUS :: a(:), b(:), c(:)
  !$omp simd
  DO i=1,n
    c(i) = a(i) + b(i)
  END DO
END SUBROUTINE ssum
```



Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP SIMD 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

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP SIMD clauses

- **safelen (*length*)**
  - Maximum number of iterations that can run concurrently without breaking a dependence (maximum vector length in practice)
- **linear (*linlist*[:*linear-step*])**
  - ***linlist*** is either ***list*** or **modifier*(list)*** with **modifier=*ref*|*val*|*uval***
  - Defines variable's value is in relationship with the iteration number

    $x_i = x_{orig} + i * \text{linear-step}$
- **aligned (*list*[:*alignment*])**
  - Specifies a given alignment for the list items
  - The default alignment is that of the architecture
- **collapse(*n*)**
  - Combine the iteration space of the next **n** loops

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP DO SIMD 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
```

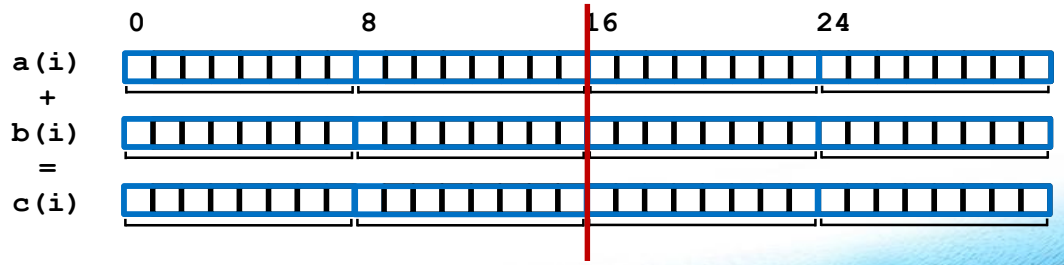
- Syntax (Fortran)

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


Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP DO SIMD: example

```
SUBROUTINE ssum(n, a, b, c)
  INTEGER :: n, i
  REAL(KIND=real64), POINTER, CONTIGUOUS :: a(:), b(:), c(:)
  !$omp do simd
  DO i=1,n
    c(i) = a(i) + b(i)
  END DO
END SUBROUTINE ssum
```



Copyright © 2018 Intel Corporation. All rights reserved.

SIMD function vectorization

- Declare one or more functions or subroutines 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):

```
!$omp declare simd                ! Within function body
!$omp declare simd(proc-name-list) ! At call site
```


Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `DECLARE SIMD`: example

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

```
REAL FUNCTION func(x, xp)
  !$omp declare simd(func) uniform( xp )
  REAL :: x, xp, denom
  denom = (x-xp)**2
  func = 1./sqrt(denom)
END FUNCTION
!$omp simd  private(x) reduction(+:sumx)
DO i = 1, nx-1
  x = x0 + i * h
  sumx = sumx + func(x, xp)
END DO
```

```
remark #15347: FUNCTION WAS VECTORIZED with...
```

`xp` is constant, `x` can be a vector

These clauses are required for correctness, just like with OpenMP threading

```
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
remark #15484: vector function calls: 1
```

SOFTWARE AND SERVICES

SIMD function must have an explicit interface

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `DECLARE SIMD`: example

- Generate a SIMD-enabled (vector) version of a scalar subroutine that can be called from a vectorized loop:

```
SUBROUTINE compute(x, y)
!$omp declare simd(compute) linear(ref(x, y))
  real, intent(in)  :: x
  real, intent(out) :: y
  y = 1. + sin(x)**3
END SUBROUTINE compute
…
!$omp simd
DO j = 1,n
  CALL compute(a(j), b(j))
END DO
```

```
remark #15347: FUNCTION WAS VECTORIZED with...
```

Important because arguments are passed by reference in Fortran

```
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
remark #15484: vector function calls: 1
```

SOFTWARE AND SERVICES

SIMD subroutine must have an explicit interface

Copyright © 2018 Intel Corporation. All rights reserved.

SIMD function vectorization clauses

- **`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)`**

Copyright © 2018 Intel Corporation. All rights reserved.

SIMD function arguments and `LINEAR(REF)`

- Whenever SIMD function arguments are passed by reference:
  - The compiler places consecutive addresses in a vector register, resulting in a gather from the addresses when the values are needed (**=slow**)
  - `LINEAR(REF(…))` tells the compiler that the addresses are consecutive, resulting to a single dereference and then copy of the consecutive values to a vector register (**=fast**)
- Recall that Fortran passes all arguments by reference
  - `LINEAR(REF(…))` is **very important** for efficient SIMD vectorization of Fortran functions and subroutines

Copyright © 2018 Intel Corporation. All rights reserved.

Targeting SIMD functions for CPU ISA

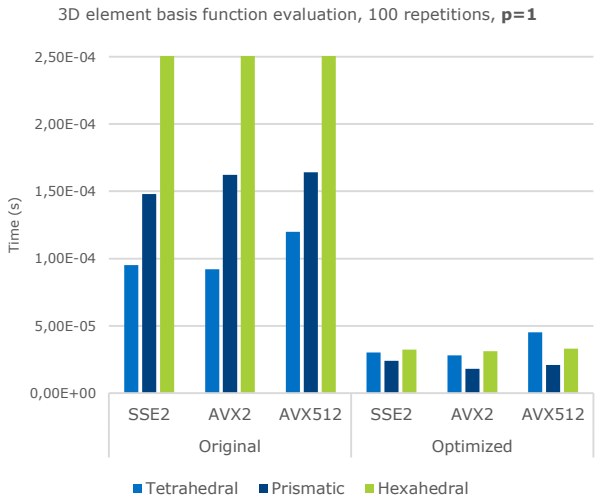
- The default binary ABI requires passing arguments in 128 bit **`xmm`** registers
  - ABI is selected irrespective of **`-xCORE-AVX2`** or **`-xCORE-AVX512`** feature flags
  - Results in inefficient 128 bit code instead of 256 or 512 bit
  - Compiler optimization report:
    `remark #15347: FUNCTION WAS VECTORIZED with xmm, simdlen=4,…`
- Intel® Fortran compiler flag **`-vecabi=cmdtarget`**
  - SIMD register width chosen according to the **`-x<feature>`**
  - Compiler optimization report:
    `remark #15347: FUNCTION WAS VECTORIZED with zmm, simdlen=16, …`

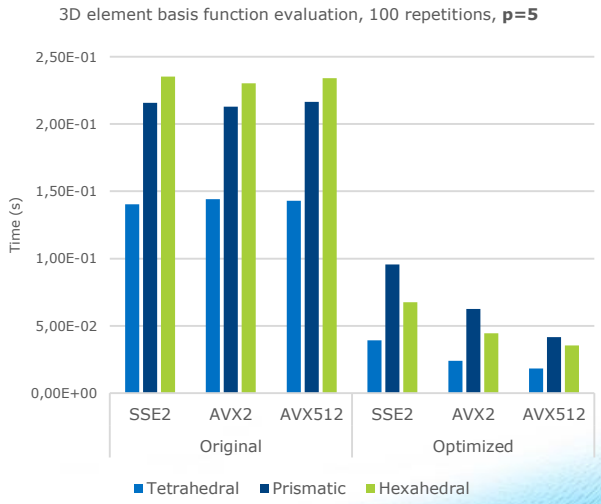
Copyright © 2018 Intel Corporation. All rights reserved.

Example: OpenMP 4.0 SIMD in Elmer

2S Intel® Xeon® Gold 6148

Results from paper: Byckling, M., Kataja, J., Klemm, M. and Zwinger, T., 2017, September. OpenMP* SIMD Vectorization and Threading of the Elmer Finite Element Software. In International Workshop on OpenMP (pp. 123-137). Springer, Cham.





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. For configuration info, see System Setup.

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

Contents

- Vectorization with Fortran array syntax
- OpenMP* SIMD
  - OpenMP* `SIMD` construct
  - OpenMP* `DECLARE SIMD` construct
- SIMD programming patterns
  - Reduction, outer loop vectorization, compress, search and histogram loops
- Summary

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

SIMD programming patterns

- Dependencies can make vectorization unsafe
- Some special patterns can still be handled by the compiler
  - The compiler may recognize a pattern (auto-vectorization)
    - Often works only for simple, 'clean' examples
  - The compiler is enforced (explicit vector programming)
    - May work for more complex cases
  - Examples: reduction, compress/expand, search, etc.
- Speed-up can come from vectorizing the rest of a large loop more than from vectorization of the pattern itself

Copyright © 2018 Intel Corporation. All rights reserved.

Reduction

```
real function reduce(n, arr)
  implicit none
  integer :: n, i
  real :: arr(n), sum
  sum = 0.0

  do i=1,n
     if (arr(i)>0) sum=sum+arr(i) ! sum causes a dependency
  end do
  reduce = sum
end function reduce
```

```
> ifort -xCORE-AVX512 -qopt-report=5 -qopt-report-file=stdout \
  -c reduce.F90 -o reduce
…
LOOP BEGIN at reduce.F90(6,3)
…
remark #15300: LOOP WAS VECTORIZED
…
```

- Reduction operations commonly auto-vectorize with any instruction set

Copyright © 2018 Intel Corporation. All rights reserved.

Reduction and floating point models

```
real function reduce(n, arr)
  implicit none
  integer :: n, i
  real :: arr(n), sum
  sum = 0.0

  do i=1,n
     if (arr(i)>0) sum=sum+arr(i) ! sum causes a dependency
  end do
  reduce = sum
end function reduce
```

```
> ifort -xCORE-AVX512 -qopt-report=5 -qopt-report-file=stdout \
  -fp-model=precise -c reduce.F90 -o reduce
…
LOOP BEGIN at reduce.F90(6,3)
   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.F90(7,20) ]
…
```

- Vectorization would change order of operations and hence the compiler is unable to vectorize

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP reductions

```
real function reduce(n, arr)
  implicit none
  integer :: n, i
  real :: arr(n), sum
  sum = 0.0
  !$omp simd reduction(+:sum)
  do i=1,n
     if (arr(i)>0) sum=sum+arr(i) ! sum causes a dependency
  end do
  reduce = sum
end function reduce
```

```
> ifort -xCORE-AVX512 -qopt-report=5 -qopt-report-file=stdout \
  -fp-model=precise -qopenmp -c reduce.F90 -o reduce
…
LOOP BEGIN at reduce.F90(7,3)
…
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
```

- Floating point model can be overridden with explicit vector reduction (OpenMP SIMD reduction)

Copyright © 2018 Intel Corporation. All rights reserved.

Outer loop vectorization

```
subroutine dist(pt, dis, n, nd, ptref)
  implicit none
  integer :: n, nd, ipt, j
  real    :: pt(nd,n), dis(n), ptref(nd), d
  !$omp simd private(j,d)
  do ipt=1,n
    d = 0.
    do j=1,nd
      d = d + (pt(j,ipt) - ptref(j))**2
    end do
    dis(ipt) = sqrt(d)
  end do
end subroutine dist
```

Outer loop with a large trip count **n**

Inner loop with a small trip count **nd**

```
LOOP BEGIN at dist.F90(7,3)
…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
LOOP BEGIN at dist.F90(9,6)
      remark #25460: No loop optimizations reported
LOOP END
```

- When **nd** is small (typically <8), outer loop vectorization may be profitable. Private copies of **j** and **d** needed for correctness

Copyright © 2018 Intel Corporation. All rights reserved.

Outer loop vectorization

```
subroutine dist(pt, dis, n, nd, ptref)
  implicit none
  integer :: n, nd, ipt, j
  real    :: pt(nd,n), dis(n), ptref(nd), d
  !$omp simd private(j,d)
  do ipt=1,n
    d = 0.
    do j=1,KNOWN_TRIP_COUNT
      d = d + (pt(j,ipt) - ptref(j))**2
    end do
    dis(ipt) = sqrt(d)
  end do
end subroutine dist
```

Outer loop with a large trip count `n`

Inner loop with a **compile time constant** small trip count `KNOWN_TRIP_COUNT` (for example 3)

```
LOOP BEGIN at dist.F90(7,3)
…
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
…
LOOP BEGIN at dist.F90(10,6)
remark #25436: completely unrolled by 3    (pre-vector)
LOOP END
```

- If the inner loop trip count is fixed and the compiler knows it, the inner loop can be completely unrolled

Copyright © 2018 Intel Corporation. All rights reserved.

Compress pattern

```
subroutine compress(a, b, na, nb )
  implicit none
  real,    intent(in ) :: a(na)
  real,    intent(out) :: b(*)
  integer, intent(in)  :: na
  integer, intent(out) :: nb
  integer              :: ia
  nb = 0
  do ia=1, na
     if(a(ia) > 0.) then
        nb = nb + 1    ! dependency
        b(nb) = a(ia) ! compress
     end if
  end do
end subroutine compress
```

```
> ifort -qopenmp -xCORE-AVX2 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c compress.F90 -o compress.o
…
LOOP BEGIN at compress.F90(9,3)
remark #25084: Preprocess Loopnests:      \
  Moving Out Store [ compress.F90(11,9) ]
remark #15344: loop was not vectorized:   \
  vector dependence prevents vectorization
…
```

- Compress pattern does not auto-vectorize with Intel® AVX2

Copyright © 2018 Intel Corporation. All rights reserved.

Compress pattern

```
subroutine compress(a, b, na, nb )
  implicit none
  real,    intent(in ) :: a(na)
  real,    intent(out) :: b(*)
  integer, intent(in)  :: na
  integer, intent(out) :: nb
  integer              :: ia
  nb = 0
  do ia=1, na
     if(a(ia) > 0.) then
        nb = nb + 1    ! dependency
        b(nb) = a(ia) ! compress
     end if
  end do
end subroutine compress
```

```
> ifort -qopenmp -xCORE-AVX512 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c compress.F90 -o compress.o
…
LOOP BEGIN at compress.F90(9,3)
remark #25084: Preprocess Loopnests: \
  Moving Out Store [ compress.F90(11,9) ]
…
remark #15300: LOOP WAS VECTORIZED
…
remark #15497: vector compress: 1
…
```

- Auto-vectorizes with Intel® AVX512 (**vcompressps** instruction)

Copyright © 2018 Intel Corporation. All rights reserved.

Compress pattern with OpenMP SIMD

```
subroutine compress(a, b, na1, na2, nb )
  real     :: a(na1,na2), b(*)
  integer  :: na1, na2, nb, ia1, ia2, ib
  real     :: sum
  nb = 0; ib=0
  !$omp simd private(ia1,sum)
  do ia2=1, na2
    sum = 0.0
    do ia1=1, na1
      sum = sum + a(ia1,ia2)
    end do
    !$omp ordered simd monotonic(ib)
    if(sum.gt.0.) then
      ib = ib + 1
      b(ib) = sum
    end if
    !$omp end ordered
  end do
  nb = ib
end subroutine compress
```

```
> ifort -qopenmp -xCORE-AVX512 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c compress.F90 -o compress.o
…
LOOP BEGIN at compress.F90(7,3)
   ...
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   ...
   remark #15497: vector compress: 1
   ...
```

Needed to express dependency on **ib**, code not correct otherwise as **!$omp simd** ignores dependencies

Copyright © 2018 Intel Corporation. All rights reserved.

Search loops

- A vectorizable loop must have a single exit and the iteration count must be known at the start of execution
  - Else a later iteration may have started before an earlier iteration decides the loop should be terminated
- Simple "search" loops are an exception which the compiler recognizes
  - executes special code if an exit occurs during a SIMD iteration
  - only works if no stores back to memory

Copyright © 2018 Intel Corporation. All rights reserved.

Search pattern (simple)

```
integer function search(na, target, array)
  implicit none
  integer, intent(in) :: na, target, array(na)
  integer :: i

  do i=1,na
     if(array(i) == target) exit
  end do

  search = i
end function search
```

```
…
LOOP BEGIN at search.F90(6,3)
…
remark #15300: LOOP WAS VECTORIZED
…
```

- Search pattern auto-vectorizes if it contains no stores back to memory

Copyright © 2018 Intel Corporation. All rights reserved.

Search pattern (with stores)

```
integer function search(a,b,c,n)
  implicit none
  real, dimension(n) :: a, b, c
  integer            :: n, i

  do i=1,n
     if(a(i) .lt. 0.) exit
     c(i) = sqrt(a(i)) * b(i)
  end do

  search = i-1
end function search
```

```
LOOP BEGIN at search_store.F90(6,3)
remark #15520: loop was not vectorized: loop with multiple \
  exits cannot be vectorized unless it meets search loop   \
  idiom criteria [ search_store.F90(9,3) ]
LOOP END
```

- Search pattern with stores does not auto-vectorize

Copyright © 2018 Intel Corporation. All rights reserved.

Search pattern (with stores, vectorized)

```
integer function search(a,b,c,n)
  implicit none
  real, dimension(n) :: a, b, c
  integer            :: n, i, j

  do i=1,n
     if(a(i).lt.0.) exit
  end do
  search = i-1

  do j=1,search
     c(j) = sqrt(a(j)) * b(j)
  end do

end function search
```

```
LOOP BEGIN at search_split.F90(6,3)
…
remark #15300: LOOP WAS VECTORIZED
…
```

```
LOOP BEGIN at search_split.F90(11,3)
…
remark #15300: LOOP WAS VECTORIZED
…
```

- Splitting the loop enables vectorization with the cost of reloading **a**

Copyright © 2018 Intel Corporation. All rights reserved.

Histogram pattern

```
 subroutine histogram(n,a, b, ind)
  implicit none
  real :: a(n), b(n), ib
  integer :: n, i, ia, ind(n)

  ! Accumulate inverse to a
  do i=1,n
     ia=ind(i)
     a(ia) = a(ia)+1/b(i)
  end do
end subroutine histogram
```

```
> ifort -qopenmp -xCORE-AVX2 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c histogram.F90 -o histogram.o
…
LOOP BEGIN at histogram.F90(7,3)
remark #15344: loop was not vectorized: vector dependence \
  prevents vectorization
…
```

- Histogram pattern does not auto-vectorize with Intel® AVX2
  - Store to **a** is a scatter (indirect addressing) and **ia** can have the same value for different values of **i**
  - Vectorization with **!$omp simd** may cause incorrect results

Copyright © 2018 Intel Corporation. All rights reserved.

Histogram pattern

```
 subroutine histogram(n,a, b, ind)
  implicit none
  real :: a(n), b(n), ib
  integer :: n, i, ia, ind(n)

  ! Accumulate inverse to a
  do i=1,n
     ia=ind(i)
     a(ia) = a(ia)+1/b(i)
  end do
end subroutine histogram
```

```
> ifort -qopenmp -xCORE-AVX512 \
  -qopt-report=5 -qopt-report-file=stdout \
  -c histogram.F90 -o histogram.o
…
LOOP BEGIN at histogram.F90(7,3)
…
remark #15300: LOOP WAS VECTORIZED
…
remark #15499: histogram: 1
```

- Histogram pattern auto-vectorizes with Intel® AVX512
  - The **VPCONFLICT** instruction detects elements with conflicting indexes, allowing the generationg of a mask for the conflict free subset of elements
  - Then re-execute the computation for remaining elements recursively

Copyright © 2018 Intel Corporation. All rights reserved.

Histogramming speed-up

- Speed-up depends on the problem details
  - Comes mostly from vectorization of other heavy computation in the loop, not from the scatter itself
  - Speed-up may be (much) less if there are many conflicts, for instance for histograms with a singularity or a narrow spike
  - Speed-up due to vectorization would be considerably higher on Intel® Xeon Phi™ x200 processors because scalar processor is slower.
- Many problems map to histograms
  - E.g. energy deposition in cells in particle transport Monte Carlo simulation

Copyright © 2018 Intel Corporation. All rights reserved.

Summary

- With Intel® Xeon processors, vectorization (and multithreading) are the keys to good floating point performance
- Application may have to be modified to improve vectorization (and threading) properties
- OpenMP 4.0 is a standardized way to program vectorized and multithreaded programs
- The compiler recognizes many common SIMD patterns

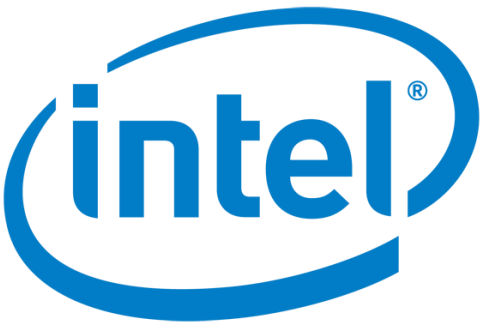
Copyright © 2018 Intel Corporation. All rights reserved.

Backup

Configuration details

Benchmarks computed on Intel internal system with Intel OPA.
Intel® Xeon® processor Gold 6148: Dual Intel® Xeon® processor Gold 6148 2.4Ghz, 20 cores/socket, 40 cores, 40 threads (HT and Turbo ON), DDR4 192 GB, 2666 MHz, RHEL 7.3, 1.0 TB SATA drive WD1003FZEX-00MK2A0, /proc/sys/vm/nr_hugepages=8000, Intel® Parallel Studio XE 2017 Update 4, tbbmalloc_proxy
Intel® Xeon® settings: Environment variables: KMP_AFFINITY=scatter,granularity=fine, I_MPI_FABRICS=shm, I_MPI_PIN_PROCESSOR_LIST=allcores:map=bunch

*Other names and brands may be claimed as the property of others
Copyright © 2018 Intel Corporation. All rights reserved.


intel®