INTEL® OPENMP TIPS

Presenter: Kenneth Craft

Date: 05-02-2017
Agenda

SIMD Directives
Explicit Vector Programming
Affinity
SIMD Directives

Two main new directives to

- vectorize ("SIMDize") loops
- create vector-version of a routine ("SIMD enabled function")
SIMD loops: syntax

#pragma omp simd [clauses]
   for-loop

!$omp simd [clauses]
   do-loops

[$!omp end simd]

Loop has to be in “Canonical loop form”
- as do/for worksharing
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 value is in relationship with the iteration number
  - \( x_i = x_{\text{orig}} + i \times \text{linear-step} \)

**aligned** (list[:alignment])
- Specifies that the list items have a given alignment
- Default is alignment for the architecture

**private** (list)

**lastprivate** (list)

**reduction** (operator:list)

**collapse** (n)

Same as existing clauses
SIMD functions: Syntax

```c
#pragma omp declare simd [clauses]

[$pragma omp declare simd [clauses]]

function definition or declaration

!$omp declare simd (function-or-procedure-name) [clauses]
```

Instructs the compiler to

- generate a SIMD-enabled version(s) of a given function
- that a SIMD-enabled version of the function is available to use from a SIMD loop
SIMD functions: 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])

Same as before
SIMD functions example

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

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

#pragma omp parallel for simd
for (i=0; i<N; i++)
    d[i] = min(distsq(a[i], b[i]), c[i]);
```
SIMD combined Constructs

Worksharing + SIMD

```c
#pragma omp for simd [clauses]
!$omp do simd [clauses]
[$!omp end do simd]
```

- The iterations are first partitioned, and then the partitions are vectorized.

Parallel + worksharing + SIMD

```c
#pragma omp parallel for simd [clause[,] clause] ...
```

```c
!$omp parallel do simd [clause[,] clause] ...
!$omp end parallel do simd
```
Explicit SIMD (Vector) Programming

Vectorization is so important
⇒ consider explicit vector programming

Modeled on OpenMP* for threading (explicit parallel programming)

• Enables reliable vectorization of complex loops that 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

• Programmer is responsible for correctness (like OpenMP threading)
  • E.g. PRIVATE and REDUCTION clauses
  • Overrides all dependencies and cost-benefit analysis

• Now incorporated in OpenMP 4.0 ⇒ portable
  • -qopenmp or -qopenmp-simd to enable
Explicit SIMD (Vector) Programming:

Use !$OMP SIMD or #pragma omp simd with -qopenmp-simd

subroutine add(A, N, X)
  integer N, X
  real A(N)
  DO I=X+1, N
    A(I) = A(I) + A(I-X)
  ENDDO
end

loop was not vectorized:
existence of vector dependence.

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
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- COLLAPSE
- LINEAR
- SAFELEN
- ALIGNED

like OpenMP for threading
(for nested loops)
(additional induction variables)
(max iterations that can be executed concurrently)
(tells compiler about data alignment)
Example: Outer Loop Vectorization

! Calculate distance from data points to reference point
subroutine dist(pt, dis, n, nd, ptref)
  implicit none
  integer, intent(in) :: n, nd
  real, dimension(nd,n), intent(in) :: pt
  real, dimension(n), intent(out) :: dis
  real, dimension(nd), intent(in) :: ptref
  integer :: ipt, j
  real :: d

!$omp simd private(d)
  do ipt=1,n
    d = 0.
    ifdef KNOWN_TRIP_COUNT
      do j=1,MYDIM ! Defaults to 3
    #else
      do j=1,nd
    #endif
      d = d + (pt(j,ipt) - ptref(j))**2
    enddo
    dis(ipt) = sqrt(d)
  enddo
end
Outer Loop Vectorization

ifort -qopt-report-phase=loop,vec -qopt-report-file=stderr -c dist.F90

... LOOP BEGIN at dist.F90(17,3)
  remark #15542: loop was not vectorized: inner loop was already vectorized
...

LOOP BEGIN at dist.F90(24,6)
  remark #15300: LOOP WAS VECTORIZED

We can vectorize the outer loop by activating the directive

!$omp simd private(d)  using -qopenmp-simd

Each iteration must have its own “private” copy of d.

ifort -qopenmp-simd -qopt-report-phase=loop,vec -qopt-report-file=stderr
  -qopt-report-routine=dist -c dist.F90

...

LOOP BEGIN at dist.F90(17,3)
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP BEGIN at dist.F90(24,6)
  remark #25460: No loop optimizations reported
LOOP END
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.


... 
LOOP BEGIN at dist.F90(17,3)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

   LOOP BEGIN at dist.F90(22,6)
      remark #25436: completely unrolled by 3 (pre-vector)
   LOOP END
LOOP END

In this case, the outer loop can be vectorized more efficiently; SIMD may not be needed.
## Outer Loop Vectorization - performance

<table>
<thead>
<tr>
<th>Optimization Options</th>
<th>Speed-up</th>
<th>What's going on</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O1</td>
<td>1.0</td>
<td>No vectorization</td>
</tr>
<tr>
<td>-O2</td>
<td>1.1</td>
<td>Inner loop vectorization</td>
</tr>
<tr>
<td>-O2 -qopenmp-simd</td>
<td>1.7</td>
<td>Outer loop vectorization</td>
</tr>
<tr>
<td>-O2 -qopenmp-simd -DKNOWN_TRIP_COUNT</td>
<td>1.9</td>
<td>Inner loop fully unrolled</td>
</tr>
<tr>
<td>-O2 -qopenmp-simd -xcore-avx2 -DKNOWN_TRIP_COUNT</td>
<td>2.4</td>
<td>Intel® AVX2 including FMA instructions</td>
</tr>
</tbody>
</table>

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary.

The results above were obtained on a 4th Generation Intel® Core™ i7-4790 system, frequency 3.6 GHz, running Red Hat® Enterprise Linux® version 7.0 and using the Intel® Fortran Compiler version 16.0 beta.
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 directive to vectorize remainder 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 DO loop

- 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:

```fortran
real function func(x,y,xp,yp)
  !$omp declare simd (func) uniform( y, xp, yp )
  real, intent(in) :: x, y, xp, yp
  denom = (x-xp)**2 + (y-yp)**2
  func = 1./sqrt(denom)
end

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

FUNCTION WAS VECTORIZED with ...

real function func(x,y,xp,yp)
  !$omp declare simd (func) uniform( y, xp, yp )
  real, intent(in) :: x, y, xp, yp
  denom = (x-xp)**2 + (y-yp)**2
  func = 1./sqrt(denom)
end

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

SIMD LOOP WAS VECTORIZED.

SIMD-enabled function must have explicit interface
!$omp simd may not be needed in simple cases
```
Clauses for SIMD-enabled Functions

```c
#pragma omp declare simd  (C/C++)

!$OMP DECLARE SIMD  (fn_name)  (Fortran)
```

- **LINEAR (REF|VAL|UVAL)** (additional induction variables)
  - use REF(X) when vector argument is passed by reference (Fortran default)

- **UNIFORM** (argument is never vector)

- **INBRANCH / NOTINBRANCH** (will function be called conditionally?)

- **SIMDLEN** (vector length)

- **ALIGNED** (tells compiler about data alignment)

- **PROCESSOR** (tells compiler which processor to target. NOT controlled by –x... switch. Intel extension in 17.0 compiler)
  - `core_2nd_gen_avx`
  - `core_4th_gen_avx`
  - `mic_avx512, ...`
Clauses for SIMD-enabled Functions

```c
#pragma omp declare simd  
!$OMP DECLARE SIMD (fn_name)
```

- **LINEAR (REF|VAL|UVAL)** (additional induction variables)
  - use REF(X) when vector argument is passed by reference (Fortran default)
- **UNIFORM** (argument is never vector)
- **INBRANCH / NOTINBRANCH** (will function be called conditionally?)
- **SIMDLEN** (vector length)
- **ALIGNED** (tells compiler about data alignment)
- **PROCESSOR**
  - `core_2nd_gen_avx` (tells compiler which processor to target. NOT controlled by –x... switch.)
  - `core_4th_gen_avx`
  - `mic_avx512, ...` (17.0 compiler only)
Use PROCESSOR clause to get full benefit on KNL

```c
#pragma omp declare simd  uniform(y,z,xp,yp,zp)
```

remark #15347: FUNCTION WAS VECTORIZED with `xmm`, simdlen=4, **unmasked**, formal parameter types: (vector,uniform,uniform,uniform,uniform)

remark #15347: FUNCTION WAS VECTORIZED with `xmm`, simdlen=4, **masked**, formal parameter types: (vector,uniform,uniform,uniform,uniform)

- default ABI requires passing arguments in 128 bit xmm registers

```c
#pragma omp declare simd  uniform(y,z,xp,yp,zp), processor(mic-avx512), notinbranch
```

remark #15347: FUNCTION WAS VECTORIZED with `zmm`, simdlen=16, **unmasked**, formal parameter types: (vector,uniform,uniform,uniform,uniform)

- Passing arguments in zmm registers facilitates 512 bit vectorization
- Independent of `-xmic-avx512` switch
- `notinbranch` means compiler need not generate masked function version
SIMD-enabled Subroutine

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

```fortran
subroutine test_linear(x, y)
    !$omp declare simd (test_linear) linear(ref(x, y))
    real(8), intent(in) :: x
    real(8), intent(out) :: y
    y = 1. + sin(x)**3
end subroutine test_linear

... Interface ...

do j = 1, n
    call test_linear(a(j), b(j))
enddo
```

SIMD-enabled routine must have **explicit interface**!

```fortran
!$omp simd not needed in simple cases like this
```
SIMD-enabled Subroutine

The LINEAR(REF) clause is very important

• In C, compiler places consecutive argument values in a vector register
• But Fortran passes arguments by reference
  • By default compiler places consecutive addresses in a vector register
  • Leads to a gather of the 4 addresses (slow)
  • LINEAR(REF(X)) tells the compiler that the addresses are consecutive; only need to dereference once and copy consecutive values to vector register
• New in compiler version 16.0.1
• Same method could be used for C arguments passed by reference

Approx speed-up for double precision array of 1M elements

| No DECLARE SIMD | 1.0 |
| DECLARE SIMD but no LINEAR(REF) | 0.9 |
| DECLARE SIMD with LINEAR(REF) clause | 3.6 |

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. The results above were obtained on an Intel® Xeon® E7-4850 v3 system, frequency 2.2 GHz, running Red Hat® Enterprise Linux® version 7.1 and using the Intel® Fortran Compiler version 16.0.1.
Loop Optimization Summary

The importance of SIMD parallelism is increasing

• Moore’s law leads to wider vectors as well as more cores
• Don’t leave performance “on the table”
• Be ready to help the compiler to vectorize, if necessary
  • With compiler directives and hints
  • Using information from vectorization and optimization reports
  • With explicit vector programming
  • Use Intel® Advisor and/or Intel® VTune™ Amplifier XE to find the best places (hotspots) to focus your efforts
• No need to re-optimize vectorizable code for new processors
  • Typically a simple recompilation
Affinity / Placement
OpenMP Affinity

Additional clause for parallel regions: proc_bind(affinity-type)

Environment variables control the affinity settings:

- **OMP_PROC_BIND**
  e.g., export OMP_PROC_BIND="master, close ,spread"

- **OMP_PLACES**
  e.g., export OMP_PLACES="{0,1,2,3},{4,5,6,7},{8:4},{12:4}"

  OMP_PLACES=threads| cores | sockets
  - threads: place → hardware*thread*
  - cores: place → core (may have multiple threads)
  - sockets: place → socket (may have multiple cores)

  e.g. export OMP_PLACES="cores (4)"

Places are system-specific and are not defined by OpenMP
Examples: master

- For best data locality
  - select OpenMP threads in the same place as the master

Examples

- master 2

- master 4

- master 8
Example: close

- For data locality, load-balancing, and more dedicated-resources
  - select OpenMP threads near the place of the master
  - wrap around once each place has received one OpenMP thread

Examples
- close 2
- close 4
- close 16
Example: spread

- For load balancing, most dedicated hardware resources
  - spread OpenMP threads as evenly as possible among places
  - create sub-partition of the place list
    - subsequent threads will only be allocated within sub-partition

Examples
- spread 2
- spread 4
- spread 8
- spread 16
Taskgroup

Allows to logically group tasks together for

- Synchronization
- Cancellation

Solves long standing complaint of not being able to wait for a task nest
Taskgroup syntax

```c
#pragma omp taskgroup
structured-block

(!$omp taskgroup
structured-block
(!$omp end taskgroup
```

Implies a wait at the end of the region on

- all child tasks created in the taskgroup
- their descendants

Remember: `taskwait` only waits for children of the current task
Task Dependencies

Allows a more unstructured way of expressing task parallelism
Potentially allows to remove more expensive synchronizations

“Create flow graphs of computations”
Task Dependencies: syntax

New clause to task construct:

```
depend( dependence-type : list )
```

dependence-type being one of:

- in
- out/inout

Dependences are constructed in serial order based on the specified data relationships:

- in waits for previous out
- out/inout wait for previous out/inout and all previous in
- no real constraint on what the task does
- only between sibling tasks
Task Dependencies

```c
void blocked_cholesky( int NB, float *A[NB][NB] ) {
    int i, j, k;
    for (k=0; k<NB; k++) {
        #pragma omp task depend(inout:A[k][k])
        spotrf (A[k][k]) ;
        for (i=k+1; i<NB; i++) {
            #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
            strsm (A[k][k], A[k][i]);
            // update trailing submatrix
            for (i=k+1; i<NB; i++) {
                for (j=k+1; j<i; j++)
                    #pragma omp task depend(in:A[k][i],A[k][j]) depend(inout:A[j][i])
                    sgemm( A[k][i], A[k][j], A[j][i] );
                    #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
                    ssyrk (A[k][i], A[i][i]);
            }
        }
    }
}
```
Task Cancellation

Allows a more unstructured way of expressing task parallelism
Potentially allows to remove more expensive synchronizations

“Create flow graphs of computations”
Cancellation Constructs

Parallel execution cannot be aborted in OpenMP 3.1
- Code regions must always run to completion
- (or not start at all)

Cancellation in OpenMP 4.0 provides a best-effort approach to terminate OpenMP regions
- Best-effort: not guaranteed to trigger termination immediately
- Triggered “as soon as” possible

Two constructs:
- Cancellation request:  #pragma omp cancel
- Cancellation points:  #pragma omp cancellation point
cancel Construct

Syntax:
```c
#pragma omp cancel construct-type-clause [ [, ]if-clause]
!$omp cancel construct-type-clause [ [, ]if-clause]
```

Clauses:
- parallel
- sections
- for (C/C++)
- do (Fortran)
- taskgroup
- if (scalar-expression)

Semantics
- Requests cancellation of the inner-most OpenMP region of the type specified
- Lets the encountering thread/task proceed to the region
cancellation point Construct

Syntax:
#pragma omp cancellation point construct-type-clause
!$omp cancellation point construct-type-clause

Clauses:
- parallel
- sections
- for (C/C++)
- do (Fortran)
- taskgroup

Semantics
- Introduces a user-defined cancellation point
- Pre-defined cancellation points:
  - implicit/explicit barriers regions
  - cancel regions
subroutine example(n, dim)
    integer, intent(in) :: n, dim(n)
    integer :: i, s, err
    real, allocatable :: B(:)

    err = 0

!$omp parallel shared(err)
...
!$omp do private(s, B)
    do i=1, n
        allocate(B(dim(i)), stat=s)

!$omp cancellation point
    if (s .gt. 0) then
        !$omp atomic write
        err = s

!$omp cancel do
    endif

...
Ordered blocks in SIMD contexts

Semantics
- The ordered with simd clause construct specifies a structured block in the simd loop or SIMD function that will be executed in the order of the loop iterations or sequence of call to SIMD functions.

Rules
- #pragma omp ordered simd is only allowed inside a SIMD loop or SIMD-enabled function.
- #pragma omp ordered simd region must be a single-entry and single-exit code block
- The strict ordered execution is only guaranteed for the block itself
  - Execution remains weakly ordered w.r.t. to outside of the block or other ordered blocks
    - Data dependencies between statements of the same block will be correctly resolved
    - Other non-vector dependencies originating in ordered block still lead to undefined behavior
Monotonic keyword for ordered

C++
#pragma omp ordered simd monotonic([var:step]s)
    structured code block

Fortran
!$omp ordered simd monotonic([var:step]s)

Semantics
- Same as for 'omp ordered simd' with a hint that vars inside the structured block are monotonically changed with respect to execution.

Why
- With this hint compiler can generate better vector code if CPU supports compress or expand instructions.

Rules
- Explicit single* self-update for vars. *Single part can be relaxed later
- [var:step]s have similar restrictions as 'linear' clause of #pragma omp simd
- vars shouldn't be used outside of an ordered block
Overlap keyword for ordered

C++
#pragma omp ordered simd overlap(overlap_index)
structured code block

Fortran
$omp ordered simd overlap(overlap_index)

Semantics
• Same as for ‘omp ordered simd’ with a hint that overlap_index has equal values in different lanes during vector execution. Compiler will do resolving for indirect accesses w.r.t. this overlap_index.

Why
• With this hint compiler can generate better vector code if CPU supports vconflict instruction. For example, CPU has support of AVX512CD.

Rules
• Single overlap_index.

Limitations
• Bail out to ordered if mixed sizes of data types is used inside structed code block
• Compiler generates general algorithm to resolve conflicts, i.e. O(N) complexity. This will be changed, so compiler will be able to generate O(logN) algorithm for some cases.
  • O(logN) algorithm is generated if it's safe to do reassociation for conflict statement
  • O(N) algorithm is generated in all other cases.
THANK YOU!
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 © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo 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
Adapting software to make best use of KNL MCDRAM
High Bandwidth On-Package Memory API

API is open-sourced (BSD licenses)
- https://github.com/memkind; also part of XPPSL at https://software.intel.com/articles/xeon-phi-software
- User jemalloc API underneath

malloc replacement:

```c
#include <memkind.h>

hbw_check_available()
hbw_malloc, _calloc, _realloc,... (memkind_t kind, ...)
hbw_free()
hbw_posix_memalign(), _posix_memalign_psize()
hbw_get_policy(), _set_policy()

ld ... -ljemalloc -l numa -l memkind -lpthread
```
HBW API for Fortran, C++

**Fortran:**

`!DIR$ ATTRIBUTES FASTMEM :: data_object1,`
- Flat or hybrid mode only
- More Fortran data types may be supported eventually
  - Global, local, stack or heap;
  - Currently just allocatable arrays (16.0) and pointers (17.0)
  - OpenMP private copies: preview in 17.0 update 1
  - Must remember to link with libmemkind!

Possible addition in a future compiler:
- Placing FASTMEM directive before ALLOCATE statement
- Instead of ALLOCATABLE declaration

**C++:** can pass `hbw_malloc()` etc.
standard allocator replacement for e.g. STL like
```
#include <hbw_allocator.h>
std::vector<int, hbw::allocator::allocate>
```
Available already, working on documentation
HBW APIs (Fortran)

Use Fortran 2003 C-interoperability features to call memkind API

interface
  function hbw_check_available() result(avail) bind(C,name='hbw_check_available')
    use iso_c_binding
    implicit none
    integer(C_INT) :: avail
  end function hbw_check_available
end interface

integer :: istat
istat = hbw_check_available()
if (istat == 0) then
  print *, 'HBM available'
else
  print *, 'ERROR, HBM not available, return code=', istat
end if
#include <memkind.h>

```c
int hbw_get_size(int partition, size_t * total, size_t * free) {     // partition=1 for HBM
    memkind_t kind;

    int stat = memkind_get_kind_by_partition(partition, &kind);
    if(stat==0) stat = memkind_get_size(kind, total, free);
    return stat;
}
```

**Fortran interface:**
```fortran
interface
  function hbw_get_size(partition, total, free) result(istat) bind(C, name='hbw_get_size')
    use iso_c_binding
    implicit none
    integer(C_INT) :: istat
    integer(C_INT), value :: partition
    integer(C_SIZE_T) :: total, free
  end function hbw_get_size
end interface
```

HBM doesn’t show as “used” until first access after allocation
What Happens if HBW Memory is Unavailable?  (Fortran)

In 16.0: silently default over to regular memory

New Fortran intrinsic in module IFCORE in 17.0:

integer(4) FOR_GET_HBW_AVAILABILITY() returns values:

- FOR_K_HBW_NOT_INITIALIZED (= 0)
  - Automatically triggers initialization of internal variables
  - In this case, call a second time to determine availability
- FOR_K_HBW_AVAILABLE (= 1)
- FOR_K_HBW_NO_ROUTINES (= 2) e.g. because libmemkind not linked
- FOR_K_HBW_NOT_AVAILABLE (= 3)
  - does not distinguish between HBW memory not present; too little HBW available;
  - and failure to set MEMKIND_HBW_NODES

New RTL diagnostics when ALLOCATE to fast memory cannot be honored:

183/4 warning/error libmemkind not linked
185/6 warning/error HBW memory not available
Severe errors 184, 186 may be returned in STAT field of ALLOCATE statement
Controlling What Happens if HBM is Unavailable (Fortran)

In 16.0: you can’t

New Fortran intrinsic in module IFCORE in 17.0:

```fortran
integer(4) FOR_SET_FASTMEM_POLICY(new_policy)
```

input arguments:

- **FOR_FASTMEM_INFO** (= 0) return current policy unchanged
- **FOR_FASTMEM_NORETRY** (= 1) error if unavailable (default)
- **FOR_FASTMEM_RETRY_WARN** (= 2) warn if unavailable, use default memory
- **FOR_FASTMEM_RETRY** (= 3) if unavailable, silently use default memory

returns previous HBW policy

Environment variables (to be set before program execution):

- **FOR_FASTMEM_NORETRY** =T/F default False
- **FOR_FASTMEM_RETRY** =T/F default False
- **FOR_FASTMEM_RETRY_WARN** =T/F default False
Getting consistent floating-point results when moving to the Intel® Xeon Phi™ x200 processor family from Intel® Xeon® processors or from Intel® Xeon Phi™ x100 Coprocessors
Floating-Point Reproducibility

-fp-model precise  disables most value-unsafe optimizations
(especially reassociations)

- The primary way to get consistency between different platforms (including KNL)
or different optimization levels

- Does not prevent differences due to:
  - Different implementations of math functions
  - Use of fused multiply-add instructions (FMAs)

- Floating-point results on Intel® Xeon Phi™ x100 coprocessors may not be
bit-for-bit identical to results obtained on Intel® Xeon® processors or on KNL
Disabled by -fp-model precise

Vectorization of loops containing transcendental functions
Fast, approximate division and square roots
Flush-to-zero of denormals
Vectorization of reduction loops
Other reassociations
  (including hoisting invariant expressions out of loops)
Evaluation of constant expressions at compile time
...

Optimization Notice
Copyright © 2015, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Math functions

Implementation of math functions may differ between different processors

- For consistency of math functions between KNL and Intel® Xeon® processors, use `-fimf-arch-consistency=true` for both

- Not available for KNC
  - `-fp-model precise` (or `-fimf-precision=high`) should get you close

- These options come at a cost in performance
The most common cause of differences between Intel® Xeon® processors and Intel® Xeon Phi™ x100 coprocessors or KNL

- Not disabled by -fp-model precise
- Can disable for testing with -no-fma
- Or by function-wide pragma or directive:
  #pragma float_control(fma,off)
  !dir$ nofma
  - With some impact on performance
- -fp-model strict disables FMAs, amongst other things
  - But on KNC, results in non-vectorizable x87 code
- The fma() intrinsic in C should always give a result with a single rounding, even on processors with no FMA instruction
Can cause issues even when both platforms support them (e.g. Haswell and KNL)

- Optimizer may not generate them in the same places
  - No language rules
- FMAs may break the symmetry of an expression:

  \[
  c = a; \quad d = -b; \\
  \text{result} = a*b + c*d; \quad ( = 0 \quad \text{if no FMAs})
  \]

If FMAs are supported, the compiler may convert to either

\[
\text{result} = \text{fma}(c, d, (a*b)) \quad \text{or} \quad \text{result} = \text{fma}(a, b, (c*d))
\]

Because of the different roundings, these may give results that are non-zero and/or different from each other.
Reproducibility: the bottom line (for Intel64)

```
/fp:precise /Qfma- /Qimf-arch-consistency:true
   -fp-model precise -no-fma -fimf-arch-consistency=true
   (Windows*)
   (Linux* or OS X*)
   arch-consistency=true
```

- Recommended for best reproducibility
  - Also for IEEE compliance
  - And for language standards compliance (C, C++ and Fortran)

- This isn’t very intuitive
  - a single switch will do all this in the 17.0 compiler
  - -fp-model consistent  (/fp:consistent on Windows*)
Prefetching for KNL

Hardware prefetcher is more effective than for KNC
Software (compiler-generated) prefetching is off by default
  - Like for Intel® Xeon® processors
  - Enable by -qopt-prefetch=[1-5]

KNL has gather/scatter prefetch
  - Enable auto-generation to L2 with -qopt-prefetch=5
    - Along with all other types of prefetch, in addition to h/w prefetcher – careful.
  - Or hint for specific prefetches
    - !DIR$ PREFETCH var_name [: type : distance ]
    - Needs at least -qopt-prefetch=2
  - Or call intrinsic
    - _mm_prefetch((char *) &a[i], hint);    C
    - MM_PREFETCH(A, hint)                       Fortran
Gather Prefetch Example

void foo(int n, int* A, int* B, int* C)    {
    // pragma_p_f e c h t   _v a r : _h i n t : _d i s t a n c e
    #pragma prefetch A:1:3   // prefetch to L2 cache  3 iterations ahead
    #pragma vector aligned
    #pragma simd
    for(int i=0; i<n; i++)
        C[i] = A[B[i]];}

icc -O3 -xmic-avx512 -qopt-prefetch=3 -qopt-report=4 -qopt-report-file=stderr -c -S emre5.cpp

remark #25033: Number of indirect prefetches=1, dist=2
remark #25035: Number of pointer data prefetches=2, dist=8
remark #25150: Using directive-based hint=1, distance=3 for indirect memory reference [ emre5.cpp(...
remark #25540: Using gather/scatter prefetch for indirect memory reference, dist=3  [ emre5.cpp(9,12) ]
remark #25143: Inserting bound-check around lfetches for loop

% grep gatherpf emre5.s
    vgatherpf1dps (%rsi,%zmm0){%k1}                   #9.12 c7 stall 2
% grep prefetch emre5.s
    # mark_description "-O3 -xmic-avx512 -qopt-prefetch=3 -qopt-report=4 -qopt-report-file=stderr -c -S -g";
    prefetcht0 512(%r9,%rcx)                      #9.14 c1
    prefetcht0 512(%r9,%r8)                      #9.5 c7
Additional Resources (Optimization)

Webinars:
https://software.intel.com/articles/further-vectorization-features-of-the-intel-compiler-webinar-code-samples


Explicit Vector Programming in Fortran:
https://software.intel.com/articles/explicit-vector-programming-in-fortran

Initially written for Intel® Xeon Phi™ coprocessors, but also applicable elsewhere:
https://software.intel.com/articles/vectorization-essential

Compiler User Forums at  http://software.intel.com/forums
THANK YOU!
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 © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo 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