



Intel® Xeon Phi™ programming

September 22nd-23rd 2015
University of Copenhagen,
Denmark

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

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

SOFTWARE AND SERVICES

Multiple levels of parallelism

- Dimensions of parallelism
 - Across multiple applications (program)
 - Across multiple processes (process)
 - ***Across multiple threads*** (***thread***)
 - ***Across multiple instructions*** (***SIMD, or "vector"***)
- **Single Instruction Multiple Data (SIMD)**
 - Performance gains because a single instruction performs more work
 - Data parallelism



SIMD vectorization

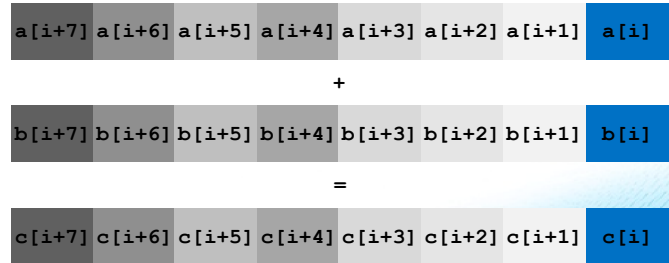
Intel® Xeon Phi™, Knights Corner (KNC)

SOFTWARE AND SERVICES

Vectorization of code

- Transform sequential code to exploit SIMD processing capabilities of Intel® processors
 - Calling a vectorized library
 - Automatically by tools like a compiler
 - Manually by explicit syntax

```
for(i = 0; i <= MAX; i++)  
    c[i] = a[i] + b[i];
```

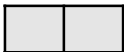


History of SIMD ISA extensions*

Intel® Pentium® processor (1993)



MMX™ (1997)



Intel® Streaming SIMD Extensions (Intel® SSE in 1999 to Intel® SSE4.2 in 2008)



Intel® Advanced Vector Extensions (Intel® AVX in 2011 and Intel® AVX2 in 2013)



Intel Many Integrated Core Architecture (Intel® MIC Architecture in 2013), Intel® AVX-512 in 2015



* Illustrated with the number of 32-bit data elements that are processed by one “packed” instruction.

Vectorization software architecture

Vector Options

Intel® Math Kernel Library

Auto vectorization

**Semi-auto vectorization:
#pragma (vector, ivdep, simd)**

**Array Notation: Fortran,
Intel® Cilk™ Plus**

**C/C++ Vector Classes
(F32vec16, F64vec8)**

OpenCL*

Intrinsics

Ease of use



Fine control

SOFTWARE AND SERVICES

Overview of vector code types

- **Auto-Vectorization**

```
for (int i = 0; i < N; ++i) {  
    A[i] = B[i] + C[i];  
}
```

- **Array notation**

```
A(:) = B(:) + C(:)
```

- **OpenMP SIMD construct**

```
#pragma omp simd  
for (int i = 0; i < N; ++i) {  
    A[i] = B[i] + C[i];  
}
```

- **OpenMP SIMD function**

```
#pragma omp declare simd  
float ef(float a, float b) {  
    return a + b;  
}  
#pragma omp simd  
for (int i = 0; i < N; ++i)  
    A[i] = ef(B[i], C[i]);
```


Automatic vectorization

- The vectorizer for Intel® MIC architecture works just like for SSE, AVX or AVX2 on the host (C/C++, Fortran)
 - Enabled by default at optimization level -O2
 - Data alignment to 64 bytes
 - Vector masks, gather/scatter instructions and fused multiply-add (FMA) enable better vectorization of code
- Vectorized loops may be recognized by
 - Compiler vectorization and optimization reports
`-qopt-report-phase=vec -qopt-report-level=n`
 - Looking at the assembly code, `-s`
 - Using Intel® VTune™ or Advisor XE

Compiler optimization report

Begin optimization report for: not_vectorizable(float *, float *, float *, int *)

Report from: Interprocedural optimizations [ipo]

INLINE REPORT: (not_vectorizable(float *, float *, float *, int *)) [1] vectorize.cc(4,63)

Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]

LOOP BEGIN at vectorize.cc(5,9)

remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details

remark #15346: vector dependence: assumed ANTI dependence between line 7 and line 7

remark #25439: unrolled with remainder by 2

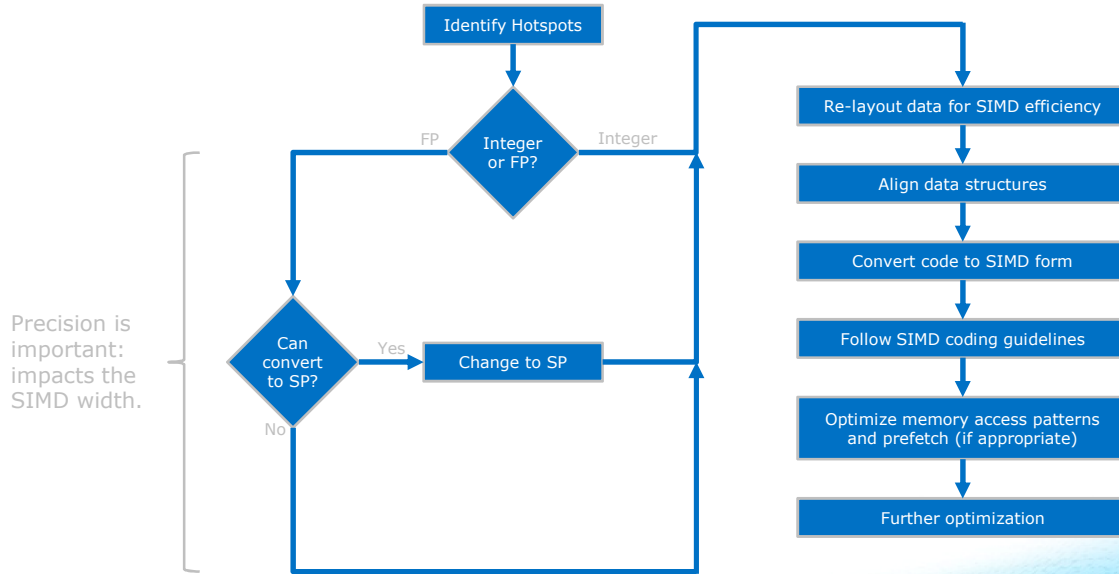
LOOP END

LOOP BEGIN at vectorize.cc(5,9)

<Remainder>

LOOP END

Preparing code for SIMD



Data Layout – why it is important

- **Instruction-Level**

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

- **Memory-Level**

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

Data layout – common layouts

Array-of-Structs (AoS)

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

- **Pros:**
Good locality of $\{x, y, z\}$,
1 memory stream
- **Cons:**
Potential for gather/scatter

Struct-of-Arrays (SoA)

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

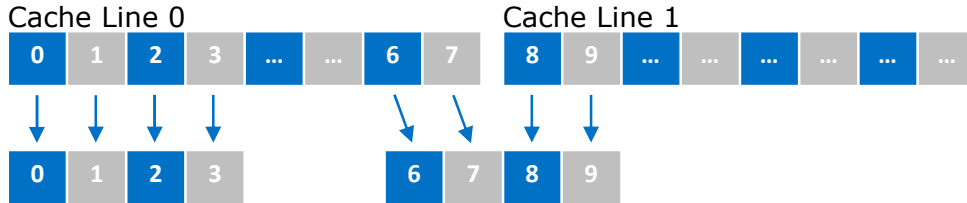
- **Pros:**
Contiguous load/store
- **Cons:**
Poor locality of $\{x, y, z\}$,
3 memory streams

Hybrid (AoSoA)

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

- **Pros:**
Contiguous load/store,
1 memory stream
- **Cons:**
Not a “normal” layout

Data alignment – why it is important



Aligned Load

- Address is aligned
- One cache line
- One instruction

Unaligned Load

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

Data alignment – sample applications

- 1) Align Memory

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

- 2) Access Memory in an Aligned Way

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

- 3) Tell the Compiler

- `#pragma vector aligned / !dir$ vector aligned`

- `__assume_aligned(p, 16) / !dir$ assume_aligned (p, 16)`

- `__assume(i % 16 == 0) / !dir$ assume (mod(i,16) .eq. 0)`

Data alignment – real-life applications

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

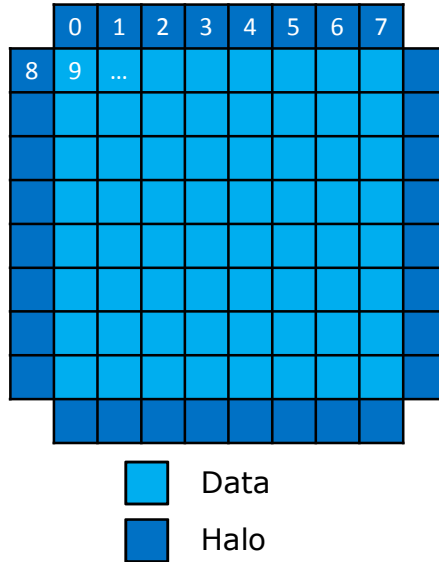
 Data

Data alignment – real-life applications

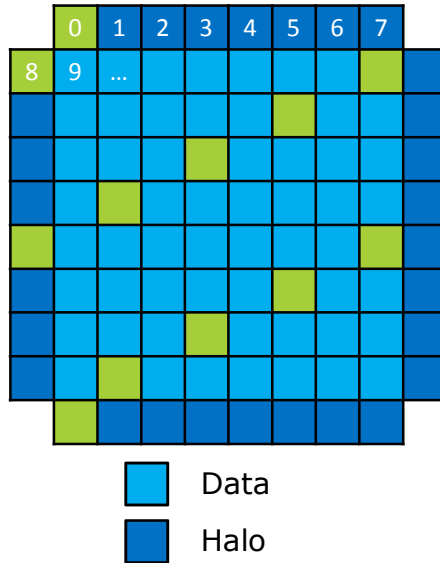
0	1	2	3	4	5	6	7
8	9	...					

 Data

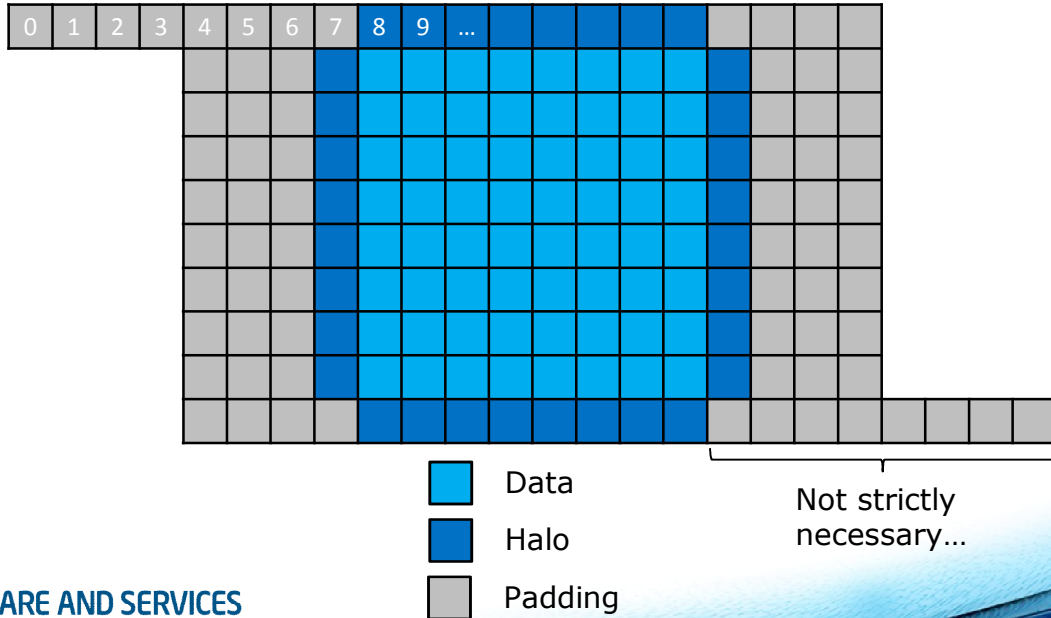
Data alignment – real-life applications



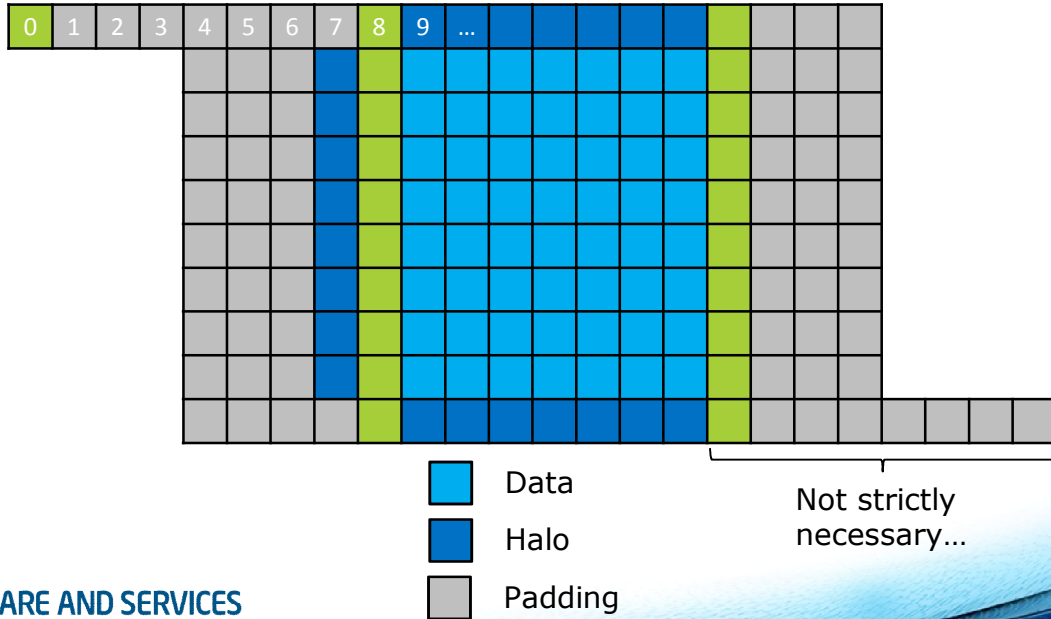
Data alignment – real-life applications



Data alignment – real-life applications



Data alignment – real-life applications





OpenMP 4.0 SIMD

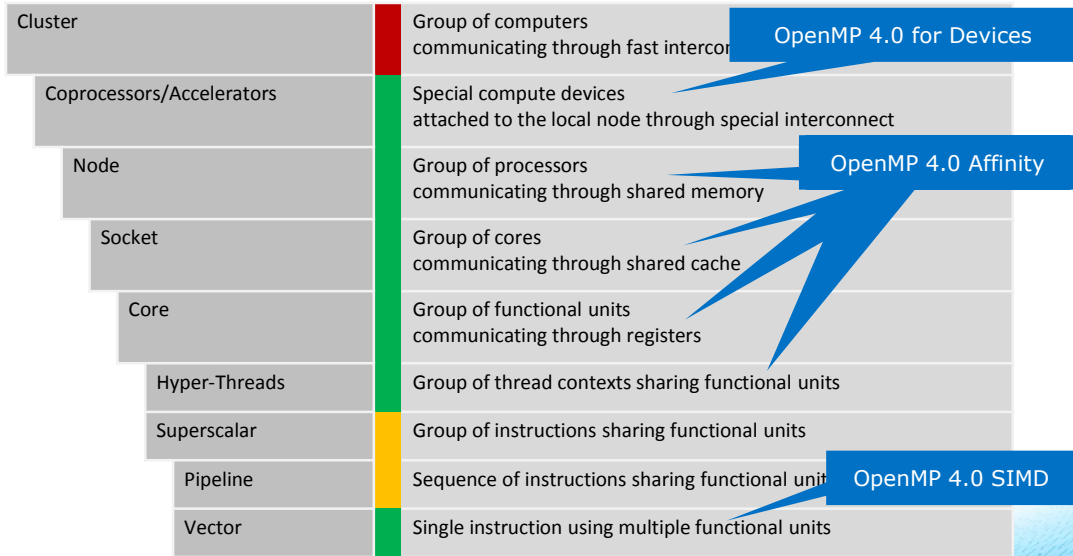
Intel® Xeon Phi™, Knights Corner (KNC)

SOFTWARE AND SERVICES

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

Levels of parallelism in OpenMP 4.0



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)

Explicit vectorization: example

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
    sum += *p;
    p += step;
}
```

- The two += operators have different meaning from each other
- The programmer should be able to express those differently
- The compiler has to generate different code
- The variables `i`, `p` and `step` have different “meaning” from each other

Explicit vectorization: example

```
#pragma omp declare simd simdlen(16)
uint32_t mandel(fcomplex c)
{
    uint32_t count = 1; fcomplex z = c;
    for (int32_t i = 0; i < max_iter; i += 1) {
        z = z * z + c; int t = cabsf(z) < 2.0f;
        count += t;
        if (!t) { break; }
    }
    return count;
}
```

- `mandel()` function is called from a loop over X/Y points
- We would like to vectorize that outer loop
- Compiler creates a vectorized function that acts on a vector of N values of c

Before OpenMP 4.0 SIMD

- Programmers had to rely on auto-vectorization...
- ... or to use vendor-specific extensions
 - Programming models (e.g., Intel® Cilk™ Plus)
 - Compiler pragmas (e.g., `#pragma vector`)
 - Low-level constructs (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

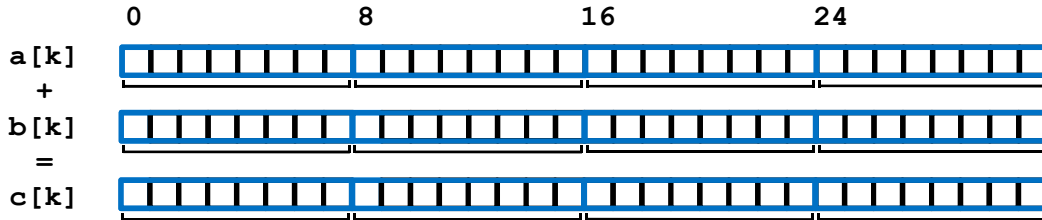
You need to trust the compiler to do the “right” thing.

OpenMP SIMD Loop Construct

- Vector *parallelism* is described 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

OpenMP SIMD: example

```
void ssum(int n, double *a, double *b, double *c) {  
#pragma omp simd  
  for (int k=0; k<n; k++)  
    c[k] = a[k] * b[k];  
}
```



OpenMP SIMD loop 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



OpenMP SIMD loop clauses

- **safelen** (*length*)
 - Maximum number of iterations that can run concurrently without breaking a dependence
 - in practice, maximum vector length
- **linear** (*list[:linear-step]*)
 - The variable's value is in relationship with the iteration number
$$X_i = X_{\text{orig}} + i * \text{linear-step}$$
- **aligned** (*list[:alignment]*)
 - Specifies that the list items have a given alignment
 - Default is alignment for the architecture
- **collapse** (*n*)
 - Combine the iteration space of the next *n* loops

OpenMP SIMD worksharing construct

- Parallelize and vectorize a loop nest
 - Distribute a loop's iteration space across a thread team
 - Subdivide loop chunks to fit a SIMD vector register

- Syntax (C/C++)

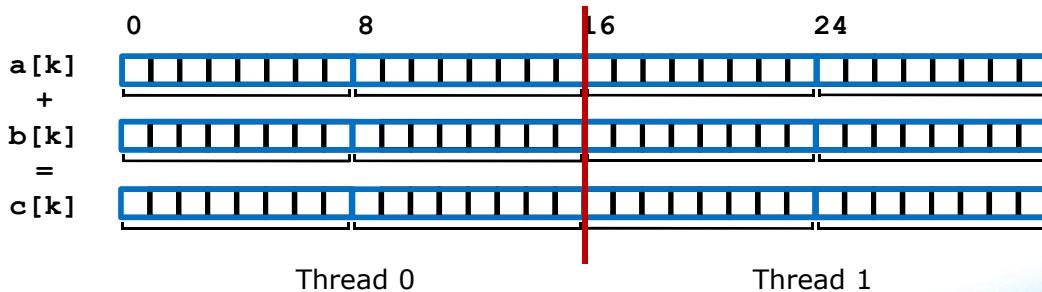
```
#pragma omp for simd [clause[[, clause],...]  
for-loops
```

- Syntax (Fortran)

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

OpenMP SIMD workshare: example

```
void ssum(int n, double *a, double *b, double *c) {  
#pragma omp for simd  
    for (int k=0; k<n; k++)  
        c[k] = a[k] * b[k];  
}
```



SIMD Function vectorization

```
float min(float a, float b) {  
    return a < b ? a : b;  
}  
  
float distsq(float x, float y) {  
    return (x - y) * (x - y);  
}  
  
void example() {  
    #pragma omp parallel for simd  
    for (i=0; i<N; i++) {  
        d[i] = min(distsq(a[i], b[i]), c[i]);  
    }  
}
```

SIMD function vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

- Syntax (C/C++):

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

- Syntax (Fortran):

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

SIMD Function vectorization

```
float min(float a, float b) {  
    return a < b ? a : b;  
}
```

```
vec16 min_v(vec16 a, vec16 b) {  
    return a < b ? a : b;  
}
```


```
float distsq(float x, float y) {  
    return (x - y) * (x - y);  
}
```

```
vec16 distsq_v(vec16 x, vec16 y) {  
    return (x - y) * (x - y);  
}
```

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

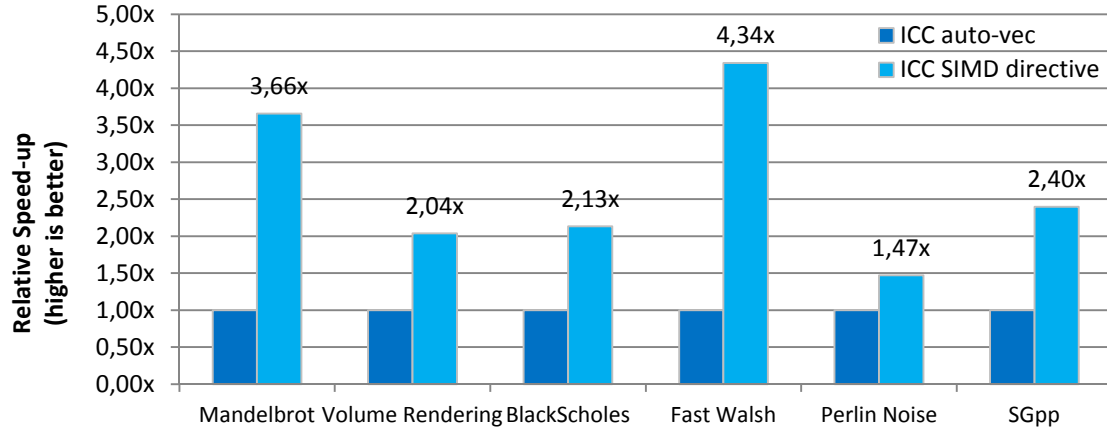
```
vd = min_v(distsq_v(va, vb, vc))
```

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)`
- 

Same as in SIMD

Explicit Vectorization – performance impact



M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, "Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP", pages 59-72, Rome, Italy, June 2012. LNCS 7312.



OpenMP threading

Intel® Xeon Phi™, Knights Corner (KNC)

SOFTWARE AND SERVICES

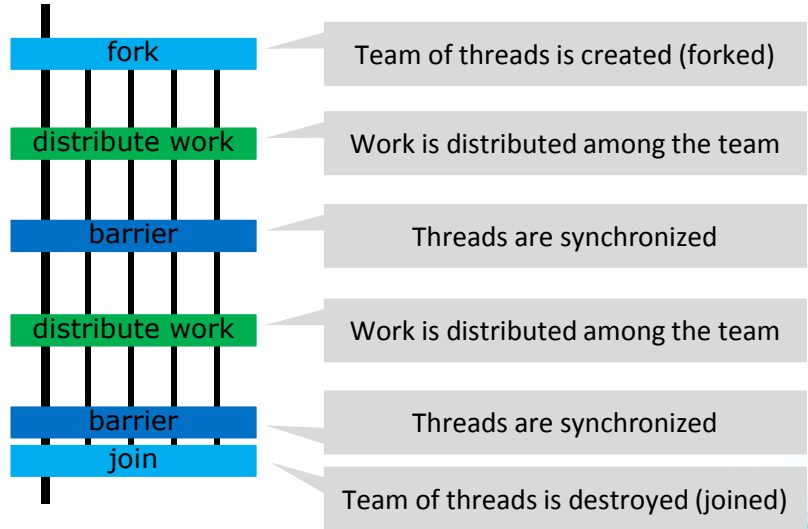
OpenMP threading in a nutshell

- In OpenMP, the description of ***parallelism*** and ***worksharing*** treated as separate entities
- Thread *parallelism* is described with `parallel` construct
C/C++: `#pragma omp parallel [clauses]`
Fortran: `!$omp parallel [clauses]`
- Loop *worksharing* is described with `loop` construct
C/C++: `#pragma omp for [clauses]`
Fortran: `!$omp do [clauses]`
- Task *worksharing* is described with `task` construct
C/C++: `#pragma omp task [clauses]`
Fortran: `!$omp task [clauses]`

OpenMP parallel: example

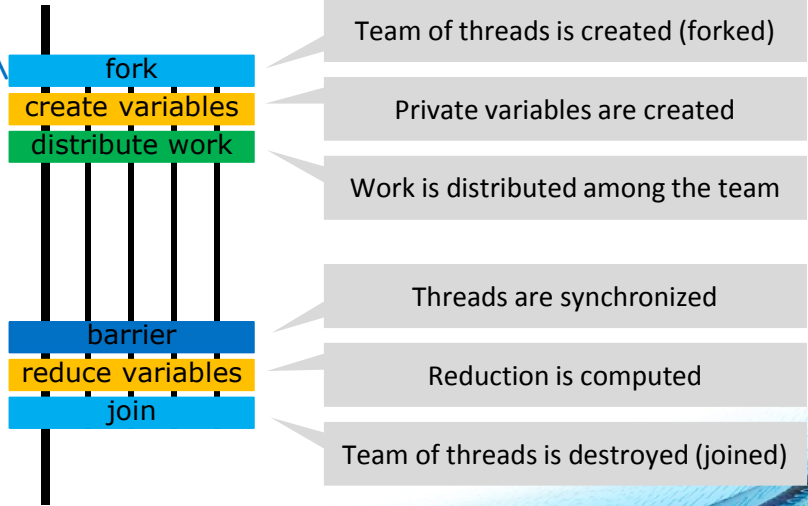
```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++)
    {...}

    #pragma omp for
    for (i = 0; i < N; i++)
    {...}
}
```



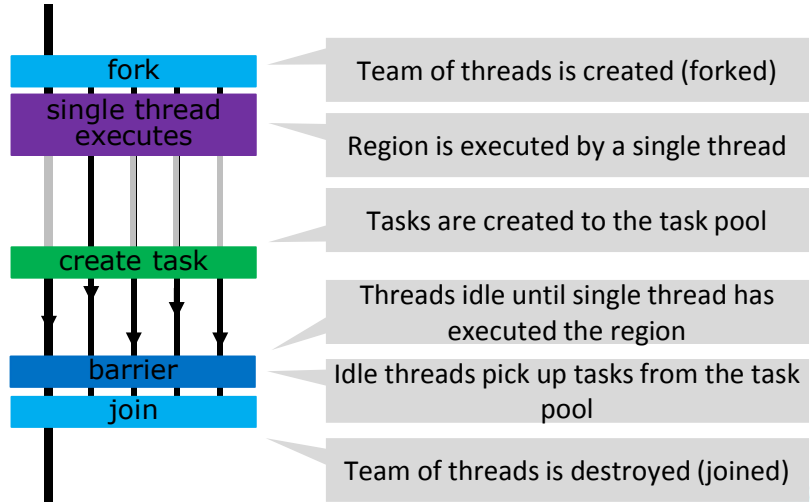
OpenMP parallel for: example

```
double a[N];
double l, s = 0.0;
#pragma omp parallel for \
  reduction(+:s) \
  private(l) \
  schedule(static,4)
for (i = 0; i<N; i++)
{
  l = log(a[i]);
  s += l;
}
```



OpenMP task: example

```
#pragma omp parallel
#pragma omp single
for(e = l->first;
    e != NULL;
    e = e->next)
#pragma omp task
{
    process(e);
}
```



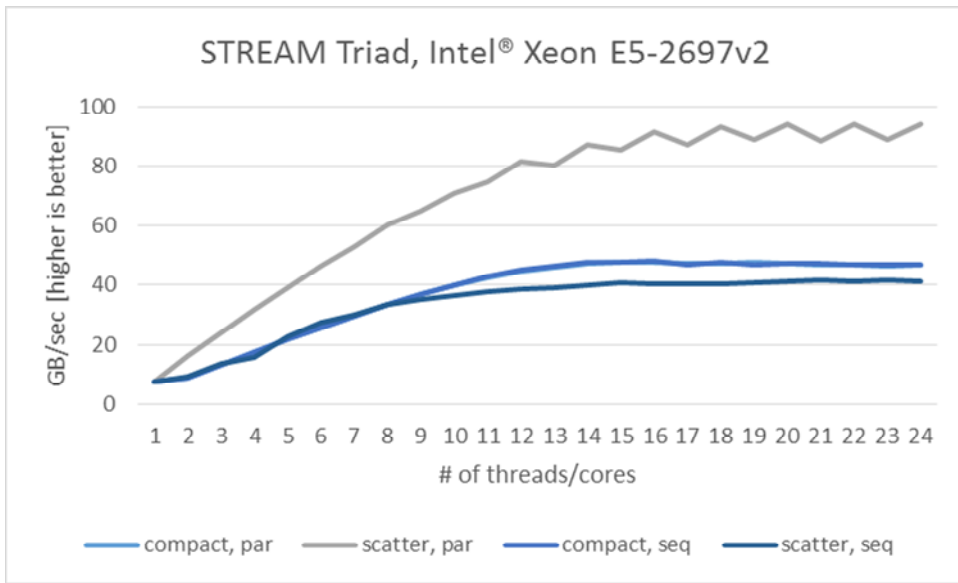
Non-uniform memory access

- Nearly all multi-socket compute servers are Non-Uniform Memory Access (NUMA) systems
 - Different access latencies for different memory locations
 - Different bandwidth observed for different memory locations

Example: Intel® Xeon E5-2600v2 Series processor



Thread affinity matters



Thread affinity – processor binding

Binding strategies depend on the machine and the application!

- Putting threads far, i.e. on different packages
 - (May) improve the aggregated memory bandwidth
 - (May) improve the combined cache size
 - (May) decrease performance of synchronization constructs
- Putting threads close together, i.e. on two adjacent cores which possible share the cache
 - (May) improve performance of synchronization constructs
 - (May) decrease the available memory bandwidth and cache size (per thread)

Thread affinity in OpenMP* 4.0

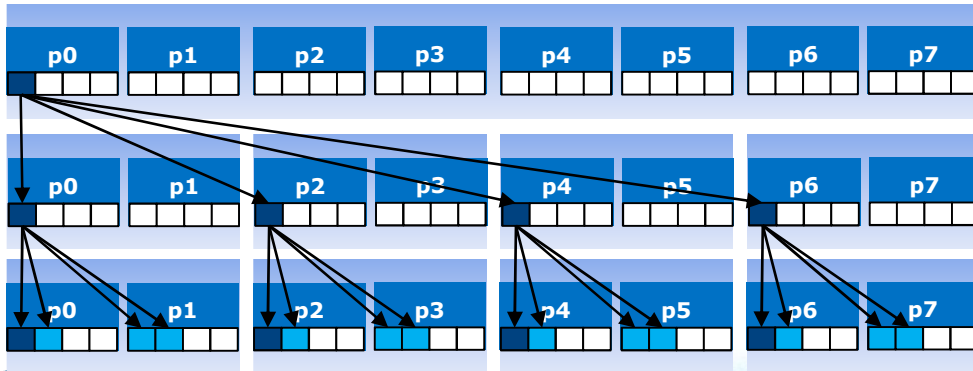
- OpenMP 4.0 introduces the concept of **places**...
 - set of threads running on one or more processors
 - can be defined by the user
 - pre-defined places available:
 - **threads** one place per hyper-thread
 - **cores** one place exists per physical core
 - **sockets** one place per processor package
- ... and affinity **policies**...
 - **spread** spread OpenMP threads evenly among the places
 - **close** pack OpenMP threads near master thread
 - **master** collocate OpenMP thread with master thread
- ... and means to control these settings
 - Environment variables **OMP_PLACES** and **OMP_PROC_BIND**
 - clause **proc_bind** for parallel regions

Thread affinity: example

- Example (Intel® Xeon Phi™):
Distribute outer region, keep inner regions close

```
OMP_PLACES=cores(8)
```

```
#pragma omp parallel proc_bind(spread)  
#pragma omp parallel proc_bind(close)
```



Conclusions

- With Intel® Xeon Phi™, vectorization and multithreading are the keys to for good 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

