OpenMP* 4.0 for HPC in a Nutshell

Dr.-Ing. Michael Klemm
Senior Application Engineer
Software and Services Group
(michael.klemm@intel.com)

*Other brands and names are the property of their respective owners.

Legal Disclaimer & Optimization Notice

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.

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, the Intel logo, Xeon, Xeon Phi, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.
*Other names and brands may be claimed as the property of others.

Optimization Notice

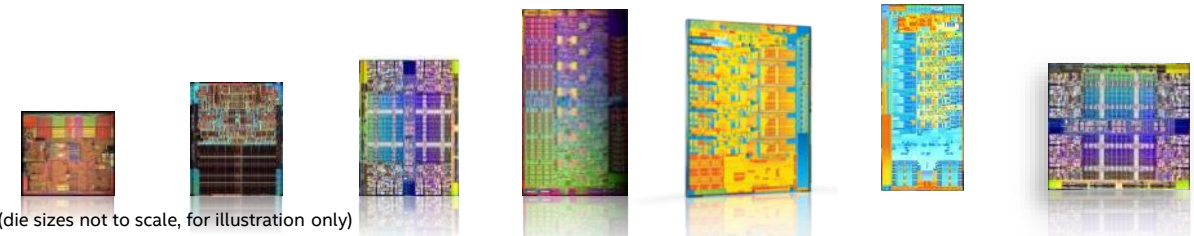
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, 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

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 (pragmas in C/C++)
- Portable across vendors and platforms
- Supports various types of parallelism

Evolution of Hardware (at Intel)



(die sizes not to scale, for illustration only)

	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® E5-2600 processor code-named Sandy Bridge EP	Intel® Xeon® E5-2600 v2 processor code-named Ivy Bridge EP	Intel® Xeon® E5-2600 v3 processor code-named Haswell EP	Intel® Xeon Phi™ coprocessor code-named Knights Corner	Intel® Xeon Phi™ processor & coprocessor code-named Knights Landing[1]
Core(s) up to	1	2	4	6	8	12	18	61	>61
Threads up to	2	2	8	12	16	24	36	244	>244
SIMD Width (bits)	128	128	128	128	256	256	256	512	512

Product specification for launched and shipped products available on ark.intel.com.
[1] Not launched or in planning.

Levels of Parallelism in OpenMP 4.0

Cluster	Group of computers communicating through fast intercon
Coprocessors/Accelerators	Special compute devices attached to the local node through special interconnect
Node	Group of processors communicating through shared memory
Socket	Group of cores communicating through shared cache
Core	Group of functional units communicating through registers
Hyper-Threads	Group of thread contexts sharing functional units
Superscalar	Group of instructions sharing functional units
Pipeline	Sequence of instructions sharing functional units
Vector	Single instruction using multiple functional units

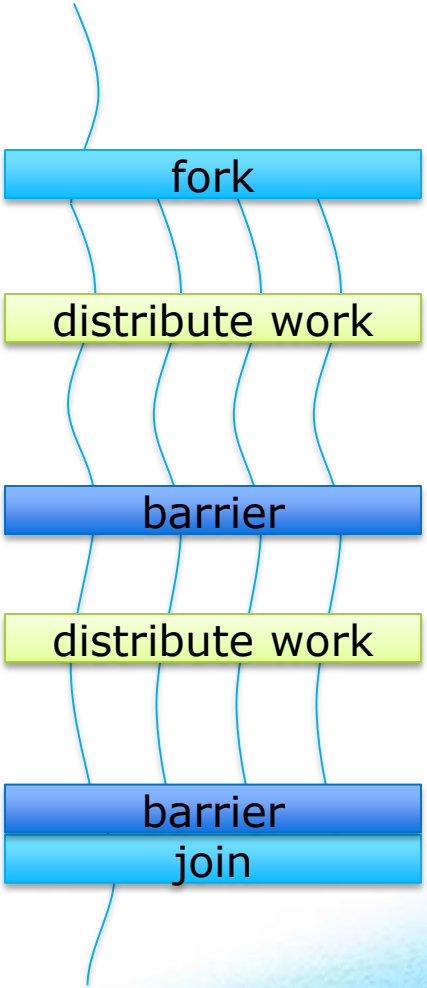
OpenMP 4.0 for Devices

OpenMP 4.0 Affinity

OpenMP 4.0 SIMD

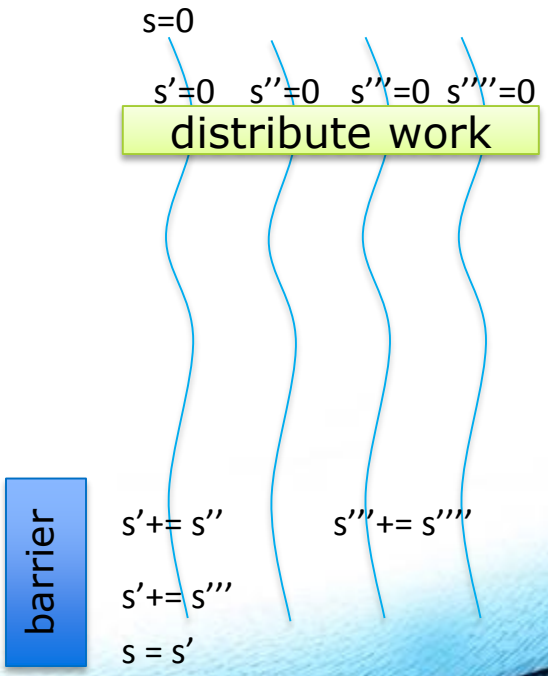
OpenMP Intro in Three Slides (1)

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

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



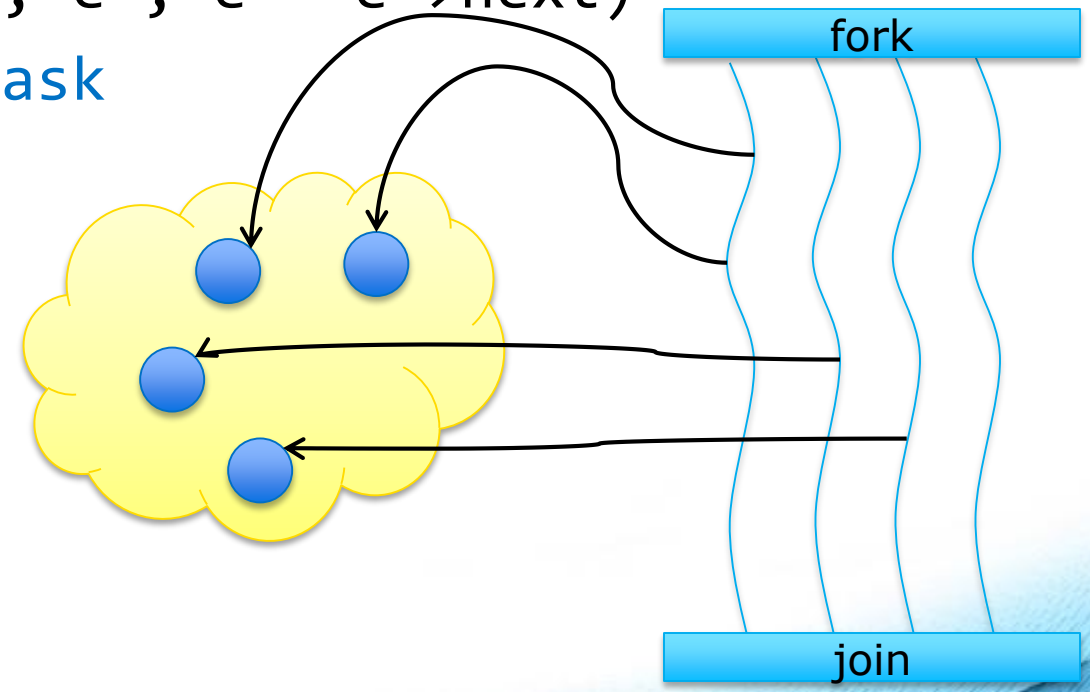
OpenMP Intro in Three Slides (2)

```
double a[N];
double l,s = 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 Intro in Three Slides (3)

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



OpenMP 4.0 SIMD

In a Time before OpenMP 4.0

- 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

- Vectorize a loop nest
  - Cut loop into chunks that fit a SIMD vector register
  - No parallelization of the loop body

- Syntax (C/C++)

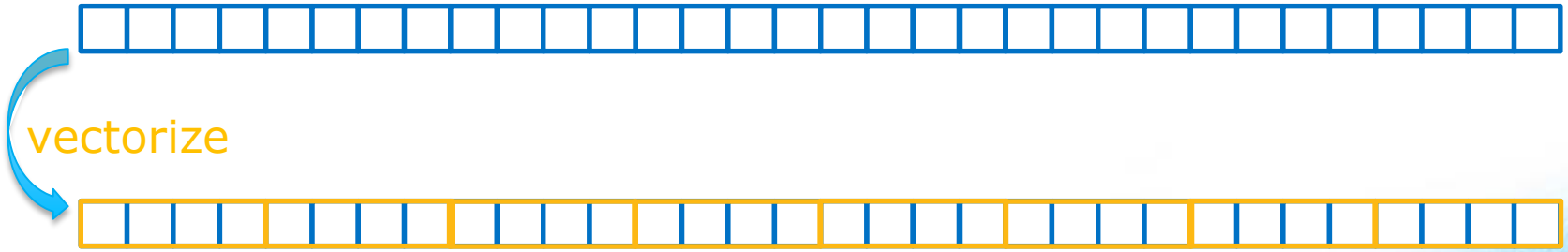
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

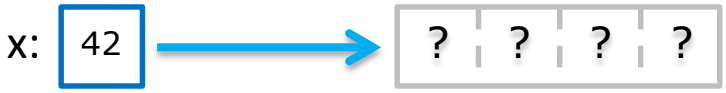
Example

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



Data Sharing 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

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

    $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (list[:alignment])`
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- `collapse (n)`

Loop-Carried Dependencies

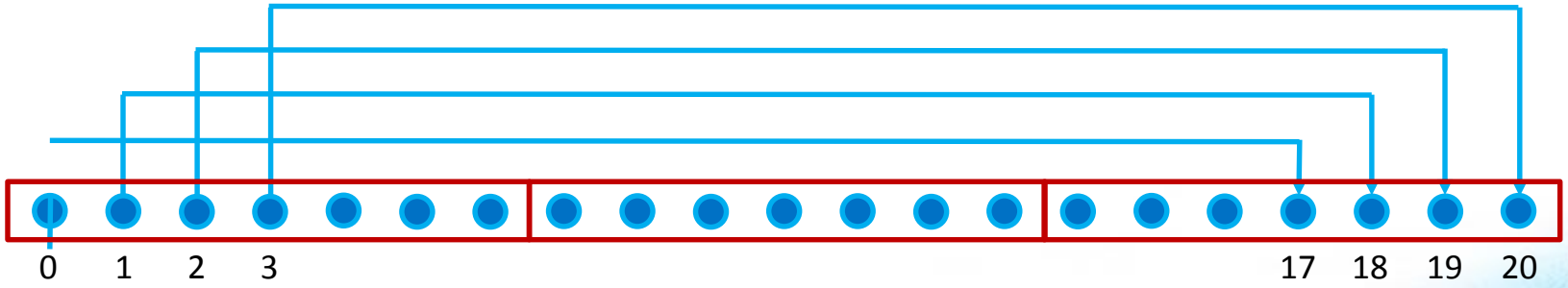
- Dependencies may occur across loop iterations
  - Loop-carried dependency
- The following code contains such a dependency:

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

- Some iterations of the loop have to complete before the next iteration can run
  - Simple trick: can you reverse the loop w/o getting wrong results?

Loop-Carried Dependencies

- Can we parallelize or vectorize the loop?
  - Parallelization: no
    (except for very specific loop schedules)
  - Vectorization: yes
    (if vector length is shorter than any distance of any dependency)



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++)

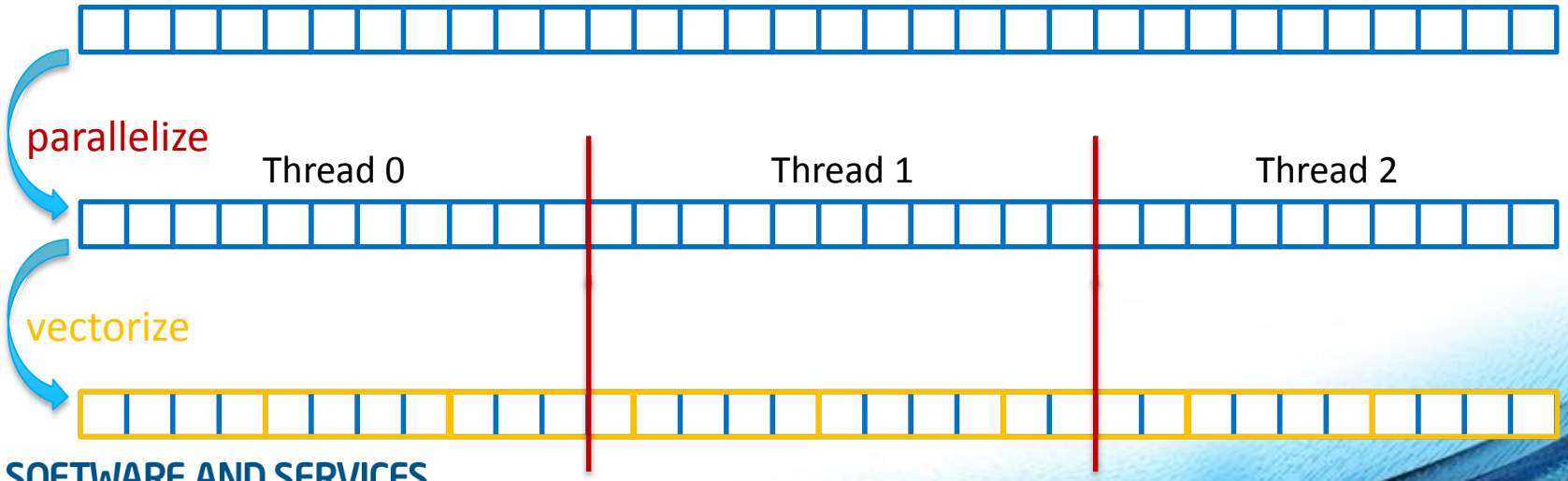
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

Example

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



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 (proc-name-list)
```

SIMD Function Vectorization

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

#pragma omp declare simd
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]);
}   }
```

```
vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}
```

```
vec8 distsq_v(vec8 x, vec8 y)
    return (x - y) * (x - y);
}
```

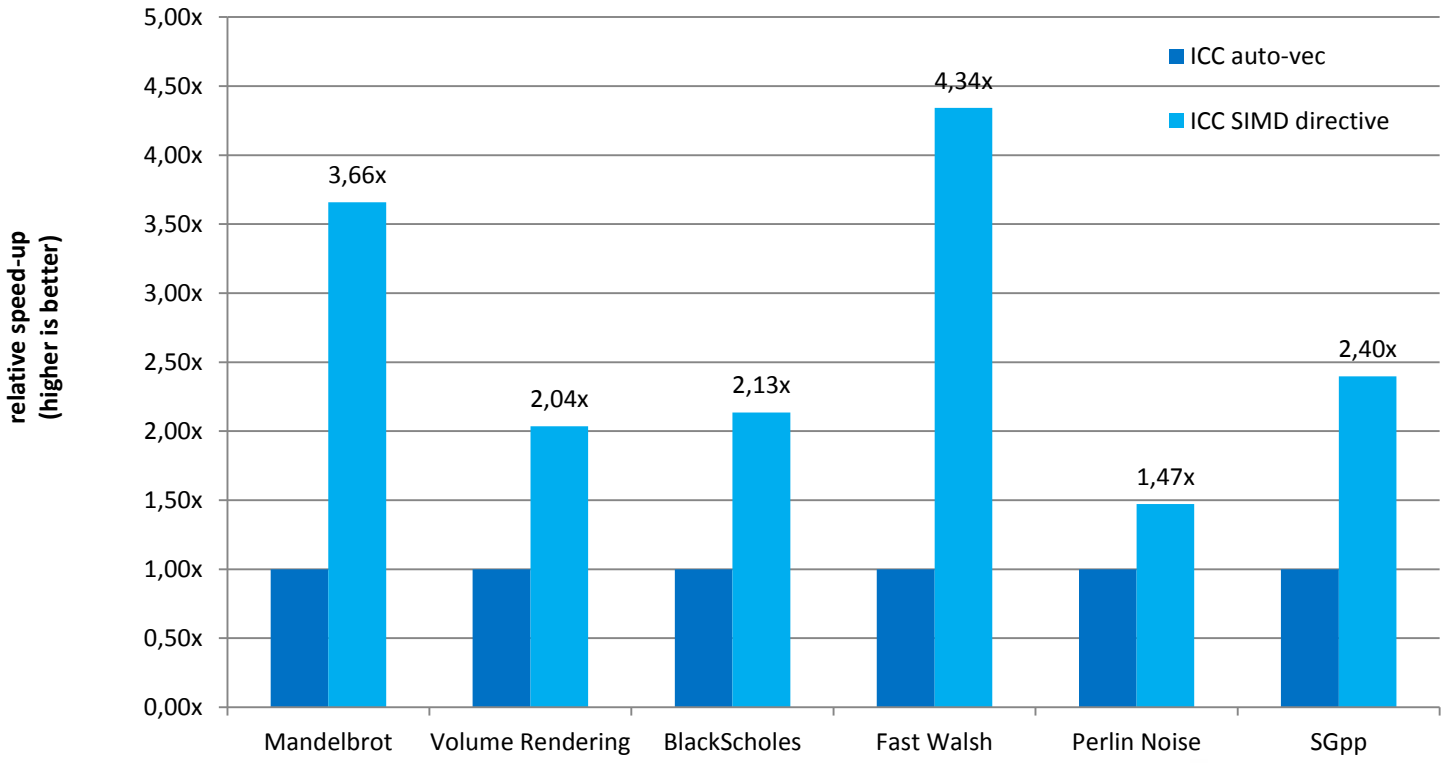
```
vd = min_v(distsq_v(va, vb, vc))
```

SIMD Function Vectorization

- `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 before

SIMD Constructs & Performance

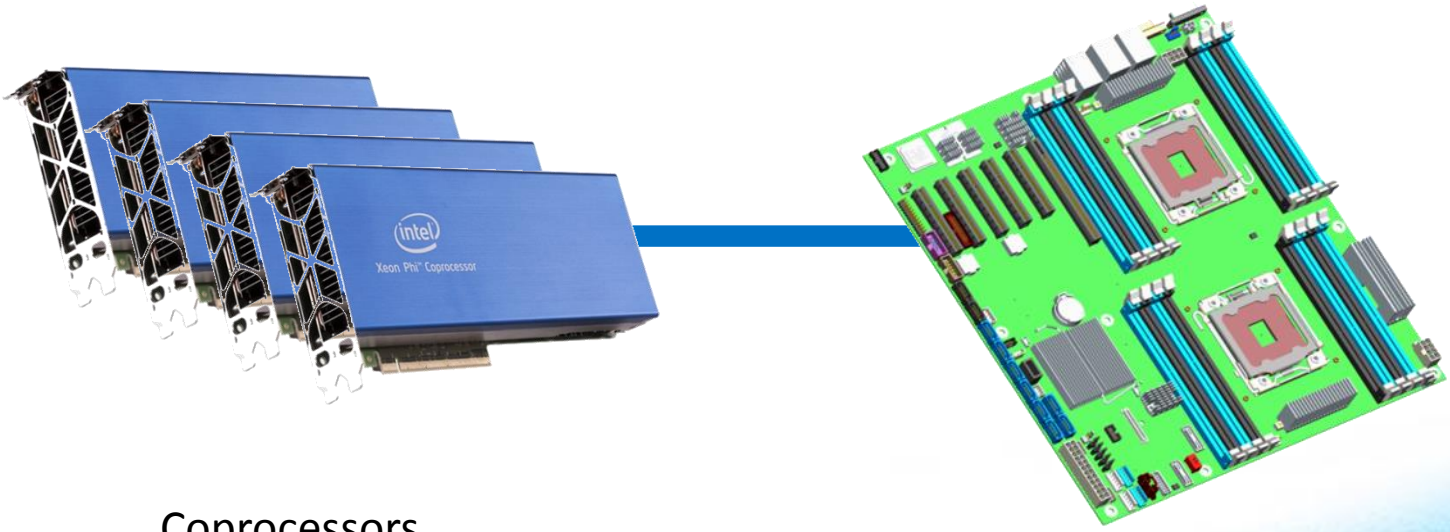


M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

OpenMP 4.0 for Devices

Device Model

- OpenMP 4.0 supports accelerators/coprocessors
- Device model:
  - One host
  - Multiple accelerators/coprocessors of the same kind



Coprocessors

Host

OpenMP 4.0 for Devices - Constructs

- Transfer control [and data] from the host to the device

- Syntax (C/C++)
```
#pragma omp target [data] [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)
```
!$omp target [data] [clause[[,] clause],…]
structured-block
!$omp end target [data]
```

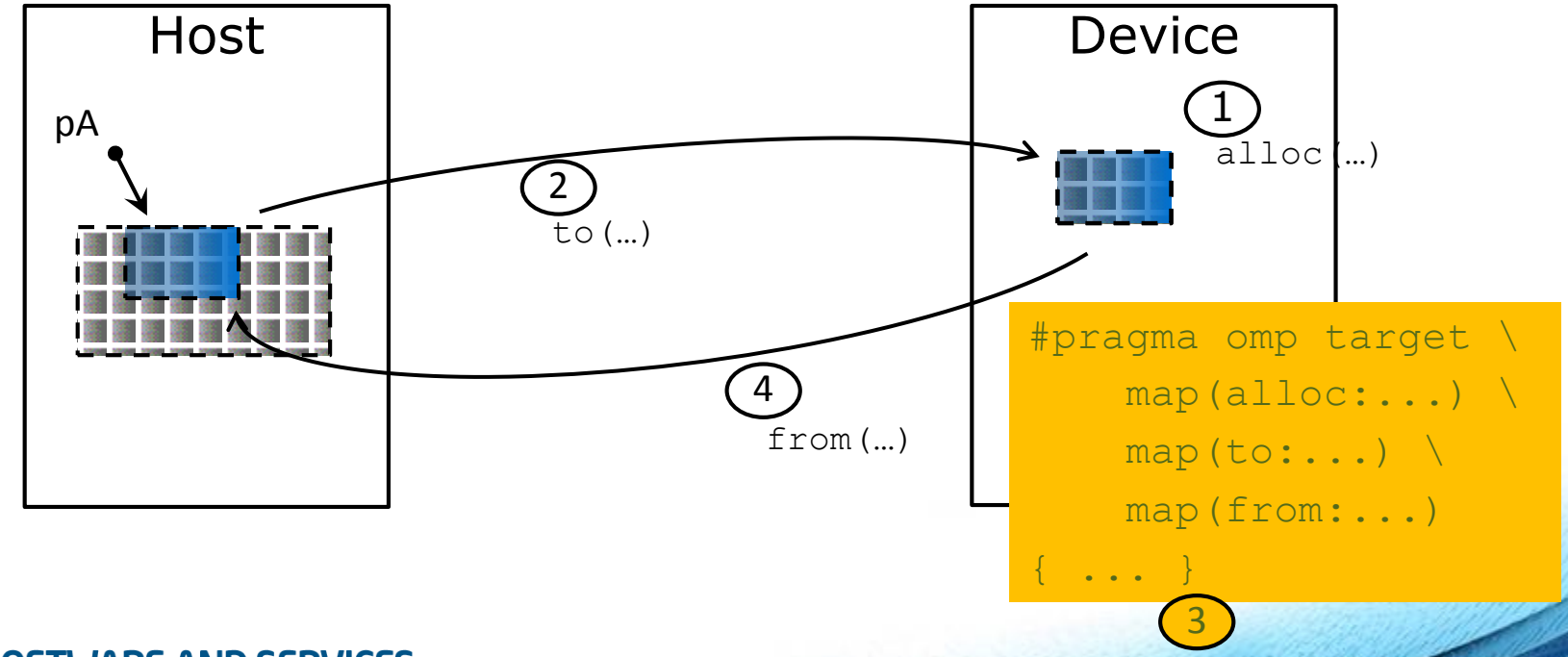
- Clauses
```
device(scalar-integer-expression)
map(alloc | to | from | tofrom: list)
if(scalar-expr)
```

Execution Model

- The `target construct` transfers the control flow to the target device
  - Transfer of control is sequential and synchronous
  - The transfer clauses control direction of data flow
  - Array notation is used to describe array length
- The `target data` construct creates a scoped device data environment
  - Does not include a transfer of control
  - The transfer clauses control direction of data flow
  - The device data environment is valid through the lifetime of the target data region
- Use `target update` to request data transfers from within a target data region

Execution Model

- Data environment is lexically scoped
  - Data environment is destroyed at closing curly brace
  - Allocated buffers/data are automatically released



Example

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
  {
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

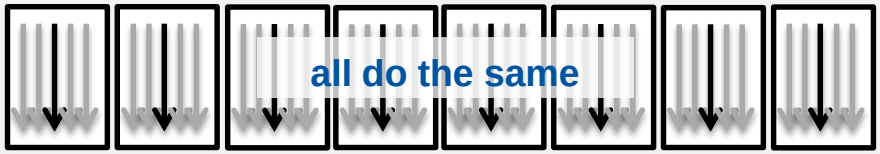
host
target
host
target
host

`teams` Construct

- Support multi-level parallel devices
- Syntax (C/C++):
  `#pragma omp teams` *`[clause[[,] clause],…]`*
  `structured-block`
- Syntax (Fortran):
  `!$omp teams` *`[clause[[,] clause],…]`*
  `structured-block`
- Clauses
  `num_teams(`*`integer-expression`*`)`
  `num_threads(`*`integer-expression`*`)`
  `default(shared | none)`
  `private(`*`list`*`), firstprivate(`*`list`*`)`
  `shared(`*`list`*`), reduction(`*`operator : list`*`)`

Offloading SAXPY to a Coprocessor

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
  {
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(nthreads)
```

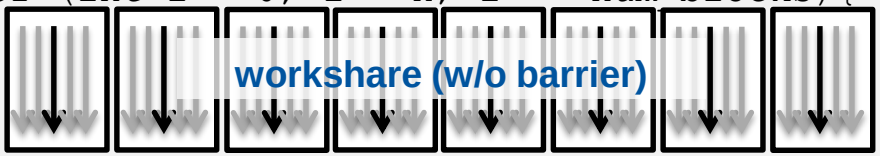



```
  for (int i = 0; i < n; i += num_blocks){
    for (int j = i; j < i + num_blocks; j++) {
        y[j] = a*x[j] + y[j];
  } }
  }
  free(x); free(y); return 0;
}
```

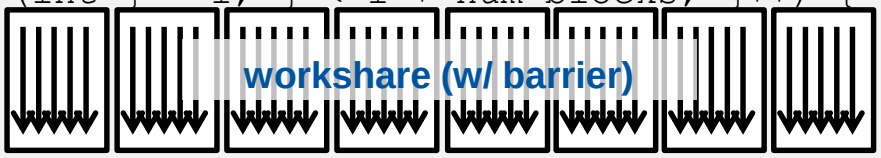
Offloading SAXPY to a Coprocessor

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
```

```
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
```




```
#pragma omp parallel for
    for (int j = i; j < i + num_blocks; j++) {
```




```
        y[j] = a*x[j] + y[j];
  } }
} free(x); free(y); return 0; }
```

Offloading SAXPY to a Coprocessor

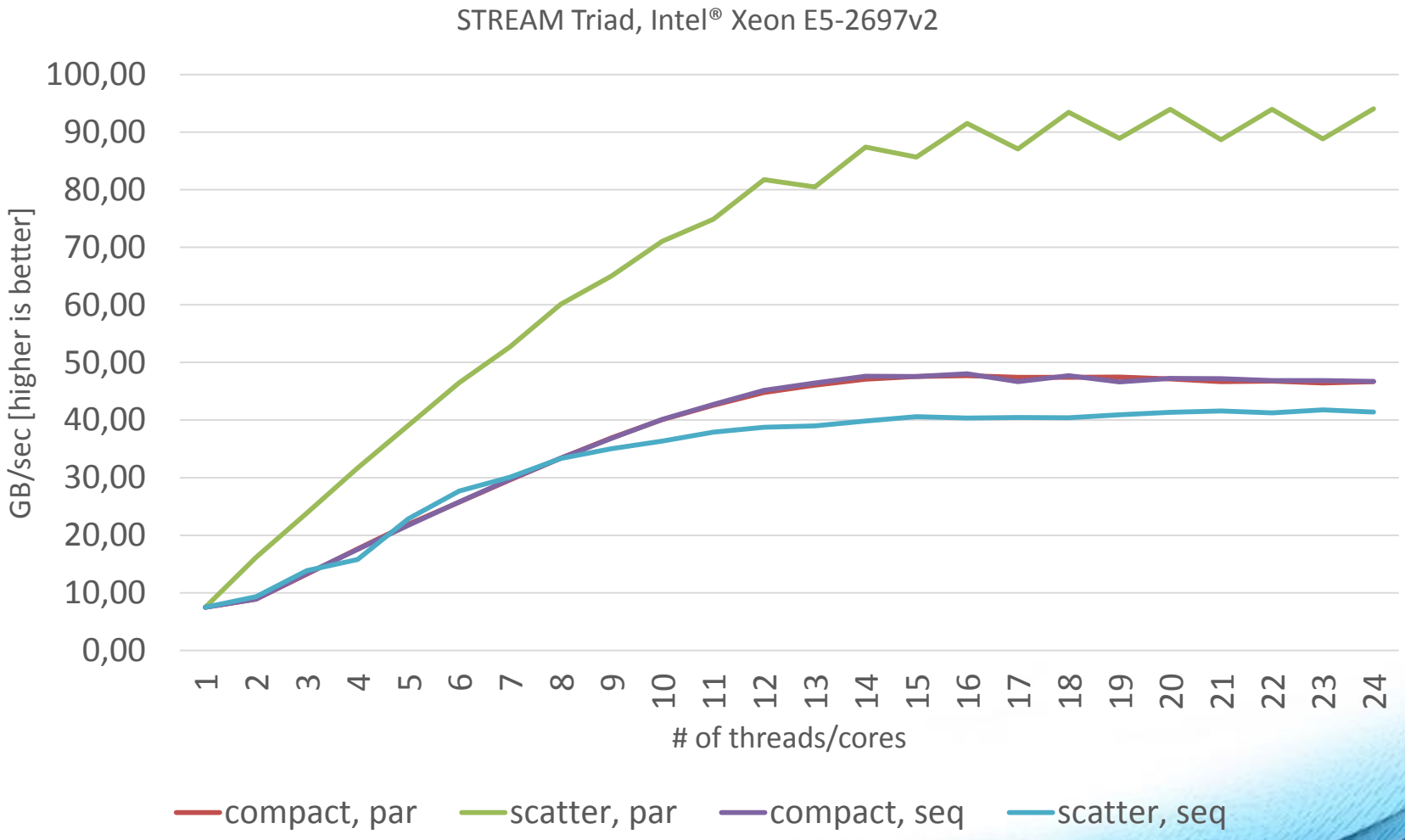
```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target map(to:x[0:n]) map(tofrom:y)
  {
#pragma omp teams distribute parallel for \
       num_teams(num_blocks) num_threads(bsize)
    for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
    }
  }

  free(x); free(y); return 0;
}
```

OpenMP 4.0 Affinity

NUMA is here to Stay…

- (Almost) all multi-socket compute servers are 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 – Why It Matters?



Thread Affinity – Processor Binding

Binding strategies depends on machine and the app

- 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…

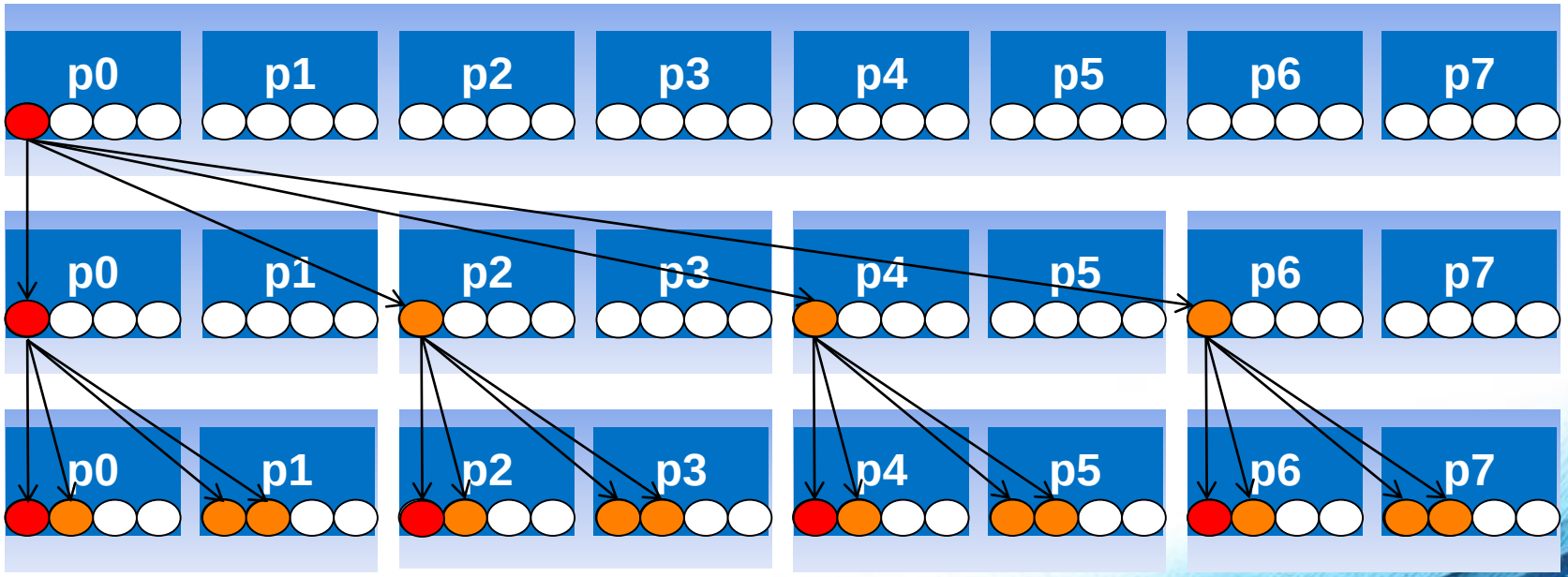
- spread spread OpenMP threads evenly among the places
- close pack OpenMP threads near master thread
- master collocate OpenMP thread with master thread

- … 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™ Coprocessor): Distribute outer region, keep inner regions close

```
OMP_PLACES=cores(8); OMP_NUM_THREADS=4,4

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

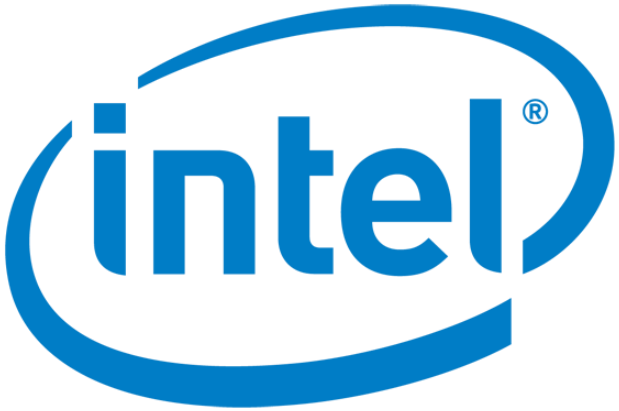


We're Almost Through

- This presentation focused on HPC stuff
- OpenMP 4.0 has more to offer!
  - Improved Fortran 2003 support
  - User-defined reductions
  - Task dependencies
  - Cancellation
- We can chat about these features during the Q&A (if you want to)

The last Slide…

- OpenMP 4.0 is a major leap for OpenMP
  - New kind of parallelism has been introduced
  - Support for heterogeneous systems with coprocessor devices
- Support in Intel® Composer XE 2013 SP1
  - SIMD Constructs (except combined constructs)
  - OpenMP for devices (except combined constructs)
  - OpenMP Affinity


intel