OpenMP* 4.0 for HPC in a Nutshell

Dr.-Ing. Michael Klemm
Senior Application Engineer
Software and Services Group

*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 © 2013 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.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

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)

Images not intended to reflect actual die sizes

<table>
<thead>
<tr>
<th></th>
<th>64-bit Intel® Xeon® processor</th>
<th>Intel® Xeon® processor 5100 series</th>
<th>Intel® Xeon® processor 5500 series</th>
<th>Intel® Xeon® processor 5600 series</th>
<th>Intel® Xeon® processor E5-2600v2 series</th>
<th>Intel® Xeon Phi™ Co-processor 7120P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Frequency</strong></td>
<td>3.6GHz</td>
<td>3.0GHz</td>
<td>3.2GHz</td>
<td>3.3GHz</td>
<td>2.7GHz</td>
<td>1.238MHz</td>
</tr>
<tr>
<td><strong>Core(s)</strong></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>12</td>
<td>61</td>
</tr>
<tr>
<td><strong>Thread(s)</strong></td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>24</td>
<td>244</td>
</tr>
<tr>
<td><strong>SIMD width</strong></td>
<td>128 (2 clock)</td>
<td>128 (1 clock)</td>
<td>128 (1 clock)</td>
<td>128 (1 clock)</td>
<td>256 (1 clock)</td>
<td>512 (1 clock)</td>
</tr>
</tbody>
</table>
# Levels of Parallelism in OpenMP 4.0

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

**OpenMP 4.0 for Devices**

**OpenMP 4.0 Affinity**

**OpenMP 4.0 SIMD**
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++)
    {
        ...
    }
    #pragma omp for
    for (i = 0; i < N; i++)
    {
        ...
    }
    #pragma omp barrier
}

SOFTWARE AND SERVICES
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;
}

SOFTWARE AND SERVICES
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);

fork
join
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()`)

```c
#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

```c
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

  \[
  \begin{array}{c}
  x: 42 \\
  \end{array} \quad \rightarrow \quad \begin{array}{c|c|c|c|c}
  \end{array}
  \]

- **firstprivate**(var-list):
  Initialized vectors for variables in var-list

  \[
  \begin{array}{c}
  x: 42 \\
  \end{array} \quad \rightarrow \quad \begin{array}{c|c|c|c|c}
  42 & 42 & 42 & 42 \\
  \end{array}
  \]

- **reduction**(op:var-list):
  Create private variables for var-list and apply reduction operator op at the end of the construct

  \[
  \begin{array}{c}
  x: 42 \\
  \end{array} \quad \rightarrow \quad \begin{array}{c|c|c|c|c}
  12 & 5 & 8 & 17 \\
  \end{array}
 \]
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
    \[ x_i = x_{\text{orig}} + i \times \text{linear-step} \]
- **aligned** (*list[::alignment]*)
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- **collapse** (*n*)
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

```c
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;
}
```

parallelize

vectorize

Thread 0 | Thread 1 | Thread 2

SOFTWARE AND SERVICES
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++):

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

• Syntax (Fortran):

```fortran
!$omp declare simd (proc-name-list)
```
SIMD Function Vectorization

```c
#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]);
    }
}
```

```c
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)`

SOFTWARE AND SERVICES
OpenMP 4.0 for Devices
Device Model

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

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

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

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

- Clauses
  
  ```c
  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)
}
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

```c
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;
}
```
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; }
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

SOFTWARE AND SERVICES
Thread Affinity – Why It Matters?

STREAM Triad, Intel® Xeon E5-2697v2

GB/sec [higher is better]

# of threads/cores

- compact, par
- scatter, par
- compact, seq
- scatter, seq

SOFTWARE AND SERVICES
Thread Affinity – Processor Binding

Binding strategies depend 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 possibly 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

```c
OMP_PLACES=cores(8)
#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