Effiziente Parallelprogrammierung mit OpenMP* 4.0

Christian Terboven <terboven@rz.rwth-aachen.de>
Michael Klemm <michael.klemm@intel.com>

Members of the OpenMP Language Committee

*Other brands and names are the property of their respective owners.
Our roles for today

Michael "the master of desaster" Klemm, works for a company that builds very fast multi-core processors and knows how to optimize codes for these. However sometimes these processors do not run programs as fast or scalable as hoped. Today, Michael has to tell us why.

Christian "the excited user" Terboven works a lot with engineers and scientists in parallelizing their codes, and was involved in the design of several OpenMP extensions. Today, Chris will explain how to use OpenMP *correctly* where some say it is not applicable.
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.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel, the Intel logo, Xeon, Core, 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 Misbeliefs

- OpenMP does not scale well
- OpenMP is only good for simple loops (Worksharing)
- OpenMP-parallel code is not elegant
- Shared Memory Parallelization is only about cores
- OpenMP does not work well for C++ codes
- For Accelerators you have to use CUDA* or OpenCL*
Agenda

- Structure of Data in Memory
- Vectorization / SIMD Parallelism
- Tasking
- Accelerators
Structure of Data in Memory
Example: Bounding Box Code

This computes a bounding box of a 2D point cloud:

```c++
struct Point2D;     /* data structure as you would expect it */
Point2D lb(RANGE, RANGE)      /* lower bound – init with max */
Point2D ub(0.0f, 0.0f);       /* upper bound – init with min */
for (std::vector<Point2D>::iterator it = points.begin();
    it != points.end(); it++) {
    Point2D &p = *it;     /* compare every point to lb, ub*/
    lb.setX(std::min(lb.getX(), p.getX()));
    lb.setY(std::min(lb.getY(), p.getY()));
    ub.setX(std::max(ub.getX(), p.getX()));
    ub.setY(std::max(ub.getY(), p.getY()));
}
```

„Problems“ for an OpenMP parallelization?

- Reduction operation has to work with non-POD datatypes
- Loop employs C++ iterator over std::vector datatype elements
Bounding Box w/ OpenMP: 😞

- The „do not“ approach:

  → Manual implementation of reduction operation with lb, ub as arrays

  ```
  Point2D *lbs = new Point2D[omp_get_max_threads()], ubs = ...
  #pragma omp parallel
  {
    lbs[omp_get_thread_num()] = Point2D(RANGE, RANGE); ubs = ...
  }

  /* in the loop: use tid = omp_get_thread_num() again */
  /* see next slide for loop body */

  #pragma omp single
  for (int t = 0; t < omp_get_num_threads(); t++) {
    lb.setX(std::min(lb.getX(), lbs[t].getX())); ...
  }
  // end omp parallel
  ```
Bounding Box w/ OpenMP: 😞

- The „do not“ approach:
  - Rewrite iterator-loop to for-loop

```cpp
int size = points.size();

#pragma omp for
for (int i = 0; i < size; i++) {
    Point2D &p = points[i];
    // tid == omp_get_thread_num()
    lbs[tid].setX(std::min(lbs[tid].getX(), p.getX()));
    lbs[tid].setY(std::min(lbs[tid].getY(), p.getY()));
    ubs[tid].setX(std::max(ubs[tid].getX(), p.getX()));
    ubs[tid].setY(std::max(ubs[tid].getY(), p.getY()));
} // end omp for
```
Except from ugly code – why does this code run so slow?
Memory Hierarchy

- In modern computer design memory is divided into different levels:
  - Registers
  - Caches
  - Main Memory

- Access follows the scheme
  - Registers whenever possible
  - Then the cache
  - At last the main memory
Cache Coherence (cc)

- If there are multiple caches not shared by all cores in the system, the system takes care of the cache coherence.

- Example:
  ```c
  int a[some_number]; // shared by all threads
  thread 1: a[0] = 23;    thread 2: a[1] = 42;
  --- thread + memory synchronization (barrier) ---
  thread 1: x = a[1];    thread 2: y = a[0];
  ```
  → Elements of array `a` are stored in continuous memory range
  → Data is loaded into cache in 64 byte blocks (cache line)
  → Both `a[0]` and `a[1]` are stored in caches of thread 1 and 2
  → After synchronization point all threads need to have the same view of (shared) main memory

- False Sharing may impact performance.
False Sharing: Parallel accesses to the same cache line may have a significant performance impact!

Caches are organized in lines of typically 64 bytes: integer array a[0-4] fits into one cache line.

Whenever one element of a cache line is updated, the whole cache line is Invalidated.

Local copies of a cache line have to be re-loaded from the main memory and the computation may have to be repeated.
Bounding Box w/ OpenMP 4.0

- **OpenMP 3.0 introduced Worksharing support for iterator loops**

  ```cpp
  #pragma omp for
  for (std::vector<Point2D>::iterator it =
       points.begin(); it != points.end(); it++) {
    ...
  }

- **OpenMP 4.0 brings user-defined reductions**

  - **name**: minp, **datatype**: Point2D
  - **read**: omp_in, **written to**: omp_out, **initialization**: omp_priv

  ```cpp
  #pragma omp declare reduction(minp : Point2D :
  omp_out.setX(std::min(omp_in.getX(), omp_out.getX())),
  omp_out.setY(std::min(omp_in.getY(), omp_out.getY()) )
  initializer(omp_priv = Point2D(RANGE, RANGE))

  #pragma omp parallel for reduction(minp:lb) reduction(maxp:ub)
  for (std::vector<Point2D>::iterator it =
       points.begin(); it != points.end(); it++) {
    ...
  }
  ```
BUT: what if the point cloud is really big?
Non-uniform Memory

- Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread

```c
double* A;
A = (double*) malloc(N * sizeof(double));
```

```c
for (int i = 0; i < N; i++) {
    A[i] = 0.0;
}
```
First Touch Memory Placement

- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition

```c
double* A;
A = (double*)
  malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
  A[i] = 0.0;
}
```
Serial vs. Parallel Initialization

- Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 („Westmere“) using Intel® Composer XE 2013 compiler with different thread binding options:

![Graph showing STREAM performance](graph.png)
Bounding Box w/ OpenMP: 😊

- Employ first-touch + thread binding in the shell:
  export OMP_PLACES=cores

as an initialization (vector always performs default initialization):

```cpp
std::valarray<Point2D> points(NUM_POINTS);
#pragma omp parallel for proc_bind(spread)
for (int i = 0; i < NUM_POINTS; i++) {
    float x = RANGE / rand();
    float y = RANGE / rand();
    points[i] = Point2D(x, y);
}
```

or with `std::vector` and appropriate allocator:

```cpp
std::vector<Point2D, no_init_allocator> points(NUM_POINTS);
```
Ok, thanks. But, how can I get even more performance from a chip?
Vectorization / SIMD Parallelism
SIMD is Here to Stay

**SSE**
Vector size: **128 bit**
Data types:
- 8, 16, 32, 64 bit integer
- 32 and 64 bit float
VL: 2, 4, 8, 16

**AVX**
Vector size: **256 bit**
Data types:
- 8, 16, 32, 64 bit integer
- 32 and 64 bit float
VL: 4, 8, 16, 32

**Intel® MIC**
Vector size: **512 bit**
Data types:
- 32 bit integer
- 32 and 64 bit float
VL: 8, 16

Illustrations: Xi, Yi & results 32 bit integer
In a Time before OpenMP 4.0...

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

- SIMD parallelism needed vendor-specific extensions
  - Programming models (e.g. C Array Notations)
  - Compiler pragmas (e.g. #pragma vector)
  - Low-level constructs

You need to trust your compiler to do the “right” thing.
SIMD vs OpenMP Threads

- SIMD parallelism does not blend with OpenMP worksharing
  "A worksharing loop has logical iterations numbered 0, 1, ..., N-1 where N is the number of loop iterations, ..."
  "The schedule clause specifies how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team."
  SIMD vectors violate these restrictions!
Exploiting SIMD Parallelism

- **Compiler:**
  - Auto-vectorization (no change of code)
  - Auto-vectorization hints (#pragma vector, ...)
  - Intel® Cilk™ Plus Array Notation Extensions

- **OpenCL**
  - (Intel® Core™/Xeon® or Intel® Xeon Phi™)
  - SIMD intrinsic class (e.g.: F32vec, F64vec, ...)
  - Vector intrinsic (e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)
  - Assembler code (e.g.: [v]addps, [v]addss, ...)

**Ease of use**
- Vendor-specific features. Often not portable across different compilers/architectures.

**Programmer control**

For Intel® C/C++ & Fortran Composer – other compilers might have similar support.

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

double pi() {
    double pi = 0.0;
    double t;

    #pragma omp simd private(t) reduction(+:pi)
    for (i=0; i<count; i++) {
        t = (double)((i+0.5)/count);
        pi += 4.0/(1.0+t*t);
    }

    pi /= count
    return pi;
}

SIMD Loop Example (More Complex)

float simd(unsigned offset, unsigned size, float *a) {
    int i;
    int sum = 0;
    float *ptr = a;

    #pragma omp simd safelen(4) reduction(+:sum) linear(ptr:1)
    for (i = 0; i < size - offset; i++) {
        a[i + offset] = *ptr; // offset = 4
        ptr += offset / 4; // Always 1 in our example
        if(a[i] > 0.0)
            sum += a[i];
        else
            sum += -1.0;
    }
    return sum;
}
SIMD Loop Clauses

- safelen(length)
  Maximum number of iterations that can run concurrently without breaking a dependence

- linear(list[:linear-step])
  The variable value is in relationship with the iteration number
  \[ x_i = x_{orig} + i \times \text{linear-step} \]

- aligned(list[:alignment])
  Specifies that the list items have a given alignment

- private(list)
- firstprivate(list)
- reduction(operator:list)
- collapse(n)

same semantics as in OpenMP 3.1
SIMDification of Functions

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

#pragma omp simd notinbranch
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]);
```
Well – these codes were all pretty simple. What about something more irregular?
Tasking
Sudoku for Lazy Computer Scientists

- Let's solve Sudoku puzzles with brute multi-core force

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

(1) Find an empty field
(2) Insert a number
(3) Check Sudoku
(4 a) If invalid:
   - Delete number,
   - Insert next number
(4 b) If valid:
   - Go to next field
### The OpenMP Task Construct

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp task [clause]</code></td>
<td><code>!$omp task [clause]</code></td>
</tr>
<tr>
<td>... structured block ...</td>
<td>... structured block ...</td>
</tr>
<tr>
<td></td>
<td><code>!$omp end task</code></td>
</tr>
</tbody>
</table>

- **Each encountering thread/task creates a new task**
  - Code and data is being packaged up
  - Tasks can be nested
    - Into another task directive
    - Into a Worksharing construct

- **Data scoping clauses:**
  - `shared(list)`
  - `private(list)`  `firstprivate(list)`
  - `default(shared | none)`
OpenMP Task Scheduling

- Default: Tasks are *tied* to the thread that first executes them → not necessarily the creator. Scheduling constraints:
  - Only the thread a task is tied to can execute it an a task can only be suspended at a suspend point (creation, finish, taskwait, barrier)
  - If task is not suspended in a barrier, executing thread can only switch to a direct descendant of all Tasks tied to the thread

- Task barrier: taskwait
  - Encountering task suspends until child tasks are complete
    - Only direct childs, not descendants!

- OpenMP barrier (implicit or explicit)
  - All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit
Parallel Brute-force Sudoku (1/3)

This parallel algorithm finds all valid solutions

1. Search an empty field
2. Insert a number
3. Check Sudoku
   - (4 a) If invalid: Delete number, Insert next number
   - (4 b) If valid: Go to next field

```c
#pragma omp parallel
#pragma omp single
such that one task starts the execution of the algorithm

first call contained in a
```
Parallel Brute-force Sudoku (2/3)

- Create an OpenMP Parallel Region team of threads
  
  ```c
  #pragma omp parallel
  {
    #pragma omp single
    solve_parallel(0, 0, sudoku2,false);
  } // end omp parallel
  ```

  → Single construct: One thread enters the execution of `solve_parallel`
  → the other threads wait at the end of the Single …
  → … and are ready to pick up threads „from the work queue“

- Syntactic sugar (you‘ll like it or you will not)
  
  ```c
  #pragma omp parallel sections
  {
    solve_parallel(0, 0, sudoku2,false);
  } // end omp parallel
  ```
Parallel Brute-force Sudoku (3/3)

- The actual implementation

```c
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
    if (!sudoku->check(x, y, i)) {
        #pragma omp task firstprivate(i,x,y,sudoku)
        {
            // create from copy constructor
            CSudokuBoard new_sudoku(*sudoku);
            new_sudoku.set(y, x, i);
            if (solve_parallel(x+1, y, &new_sudoku)) {
                new_sudoku.printBoard();
            }
        }
        // end omp task
    }
}
```

#pragma omp taskwait

#pragma omp taskwait

#pragma omp task

needs to work on a new copy of the Sudoku board

wait for all child tasks
So, how does this really work?
OpenMP Task Queues

- The runtime system keeps track of a task queue

```c
#pragma omp task   // T1
{
    do_something();
}
#pragma omp task   // T2
{
    do_something();
}
```

![Diagram of task queues](image)

- Task queue: Ta, Tb, Tc, T1, T2
OpenMP Task Queues & NUMA

- Single task queue has serious limitations
  - Single point of contention
  - NUMA effects can more easily occur
    (no locality since tasks may be scheduled to run anywhere)

- Intel Composer XE uses a distributed task queue
  - Each thread maintains its own task queue
  - Newly created tasks are added to the queue of the creator thread
  - Other threads steal tasks from other threads queues if needed
OpenMP Task Queues & NUMA

queue (thread 1)

queue (thread 2)

queue (thread 3)

queue (thread 4)

executing

Ta Tb Tc T1 T2 T3 T3
Shall I believe that?
Show me some numbers!
Performance of Task-parallel Sudoku

Sudoku on 2x Intel® Xeon® E5-2650 @2.0 GHz

Runtime [sec] for 16x16 #threads

Intel C++ 13.1, scatter binding
Give me some more power!
Accelerators / Coprocessors
Median Filter

- Example: Median filter to soften colors in an image

1. Get color values of neighboring pixels: 125, 126, 130, 123, 163, 126, 117, 115, 120
2. Sort color values (ascending): 115, 117, 120, 123, 125, 126, 126, 130, 163
3. Choose new color value (median): 115, 117, 120, 123, 125, 126, 126, 130, 163
4. Update color value
Parallel Median Filter in C/C++

```c
void MedianFilter(unsigned* inputArray, unsigned* outputArray,
                  unsigned arrayWidth, unsigned arrayHeight) {
    memset(outputArray, 0, arrayWidth * (arrayHeight+4));

#pragma omp parallel for shared(inputArray,outputArray)
    for(unsigned y = 0; y < arrayHeight; y++) { // rows
        int iOffset = (y+2) * arrayWidth;
        int iPrev = iOffset - arrayWidth;
        int iNext = iOffset + arrayWidth;
        for(unsigned  x = 0; x < arrayWidth; x++) { // columns
            unsigned uiRGBA[9];
            uiRGBA[0] = inputArray[iPrev + x - 1];
            uiRGBA[1] = inputArray[iPrev + x];
            uiRGBA[2] = inputArray[iPrev + x + 1];
            uiRGBA[3] = inputArray[iOffset + x - 1];
            uiRGBA[4] = inputArray[iOffset + x];
            // ...
        }
    }
}
```
Parallel Median Filter in C/C++

// ...

// bitonic sorting
unsigned uiMin = c4min(uiRGBA[0], uiRGBA[1]);
unsigned uiMax = c4max(uiRGBA[0], uiRGBA[1]);
uiRGBA[0] = uiMin;
uiRGBA[1] = uiMax;
uiMin = c4min(uiRGBA[3], uiRGBA[2]);
uiMax = c4max(uiRGBA[3], uiRGBA[2]);
uiRGBA[3] = uiMin;
uiRGBA[2] = uiMax;
uiMin = c4min(uiRGBA[2], uiRGBA[0]);
uiMax = c4max(uiRGBA[2], uiRGBA[0]);
// ...
// ...

uiRGBA[0] = uiMin;
uiRGBA[8] = uiMax;
uiRGBA[4] = c4max(uiRGBA[0], uiRGBA[4]);
uiRGBA[5] = c4max(uiRGBA[1], uiRGBA[5]);
uiRGBA[6] = c4max(uiRGBA[2], uiRGBA[6]);
uiRGBA[7] = c4max(uiRGBA[3], uiRGBA[7]);
uiRGBA[4] = c4min(uiRGBA[4], uiRGBA[6]);
uiRGBA[5] = c4min(uiRGBA[5], uiRGBA[7]);

// update pixel
outputArray[(y+2) * arrayWidth + x] =
c4min(uiRGBA[4], uiRGBA[5]);
} // end loop (columns)
} // end loop (rows)
__kernel void MedianFilterOpenCL(__constant uchar4* pSrc, __global uchar4* pDst) {
    const int x = get_global_id(0); const int y = get_global_id(1);
    const int width = get_global_size(0);
    const int iOffset = y * width;
    const int iPrev = iOffset - width;
    const int iNext = iOffset + width;
    uchar4 uiRGBA[9];

    // determine color values of neighboring pixels
    uiRGBA[0] = pSrc[iPrev + x - 1];
    // ...

    // bitonic sorting
    uchar4 uiMin = min(uiRGBA[0], uiRGBA[1]);
    uchar4 uiMax = max(uiRGBA[0], uiRGBA[1]);
    uiRGBA[0] = uiMin;
    uiRGBA[1] = uiMax;
    // ...
}
void ExecuteMedianFilterKernel(cl_uint* inputArray, cl_uint* outputArray, 
  cl_int arrayWidth, cl_int arrayHeight) { 
  cl_uint numStages = 0; 
  g_inputBuffer = clCreateBuffer(g_context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, 
    sizeof(cl_uint) * arrayWidth * (arrayHeight+4), inputArray, NULL); 
  g_outputBuffer = clCreateBuffer(g_context, CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR, 
    sizeof(cl_uint) * arrayWidth * (arrayHeight+4), outputArray, NULL); 
  ClSetKernelArg(g_kernel, 0, sizeof(cl_mem), (void*) &g_inputBuffer); 
  clSetKernelArg(g_kernel, 1, sizeof(cl_mem), (void*) &g_outputBuffer); 
  size_t global_work_size[2] = {arrayWidth, arrayHeight}; 
  size_t local_work_size[2] = {8, 8}; 
  size_t offset[2] = {0, 2}; 
  clEnqueueNDRangeKernel(g_cmd_queue, g_kernel, 2, offset, global_work_size, 
    local_work_size, 0, NULL, NULL); 
  clFinish(g_cmd_queue); 
  clEnqueueMapBuffer(g_cmd_queue, g_outputBuffer, true, CL_MAP_READ, 0, 
    sizeof(cl_uint) * arrayWidth * (arrayHeight+4), 0, NULL, NULL); 
  clFinish(g_cmd_queue); 
  clEnqueueUnmapMemObject(g_cmd_queue, g_outputBuffer, tmp_ptr, 0, NULL, NULL); 
  clReleaseMemObject(g_inputBuffer); g_inputBuffer = NULL; 
  clReleaseMemObject(g_outputBuffer); g_outputBuffer = NULL; 
}
Accelerator / Coprocessor Support

- OpenMP 4.0 introduces support for accelerator/coprocessor devices

- Device model:
  - One host (with traditional OpenMP threads)
  - Multiple accelerators/coprocessor (of the same kind)
#pragma omp target \
map(in:inputArray[0:arrayWidth * (arrayHeight+4)])
map(out:outputArray[0:arrayWidth * (arrayHeight+4)])
#pragma omp parallel for shared(inputArray,outputArray)
for(unsigned y = 0; y < arrayHeight; y++) { // rows
    int iOffset = (y+2) * arrayWidth;
    int iPrev = iOffset - arrayWidth;
    int iNext = iOffset + arrayWidth;
    for(unsigned  x = 0; x < arrayWidth; x++) { // columns
        unsigned uiRGBA[9];
        uiRGBA[0] = inputArray[iPrev + x - 1];
        uiRGBA[1] = inputArray[iPrev + x];
        // ...
Avoiding Unnecessary Data Transfers

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

    do_some_other_stuff_on_host();

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

- The `target` construct transfers the control flow to the target device
  - 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
  - The transfer clauses control direction of data flow
  - 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
OpenMP 4.0 Offers Even More

- Affinity control
- Task dependencies
- Taskgroups
- Cancellation of parallel execution
- Support for Fortran 2003
- Sequentially consistent atomics
Thank you for your attention!

The OpenMP 4.0 spec will (soon) be available at www.openmp.org.

As there will also be OpenMP syntax reference cards available at www.openmp.org, this presentation skipped several syntax details.