OpenMP 4.0

Feature Overview

Christian Terboven <terboven@rz.rwth-aachen.de>
11.10.2013 / ZKI AK SC, Darmstadt, Germany
Stand: 08.10.2013
Version 2.3
Agenda

- Programming for Accelerators with the target construct
- Vectorization with the simd construct
- Task Dependencies and Task Cancellation
- User-defined Reductions
- OpenMP 4.0 Feature List
Accelerators
What kind of devices shall be supported?

- **In how differs an accelerator from just another core?**
  - different functionality, i.e. optimized for something special
  - different (possibly limited) instruction set
  - → heterogeneous device

- **Assumptions used as design goals for OpenMP 4.0:**
  - every accelerator device is attached to one host device
  - it is probably heterogeneous
  - it may not be programmable in the same language as the host, or
    - it may not implement all operations available on the host
  - it may or may not share memory with the host device
  - some accelerators are specialized for loop nests
Host-centric: the execution of an OpenMP program starts on the *host device* and it may offload *target regions* to *target devices*

- In principle, a target region also begins as a single thread of execution: when a target construct is encountered, the target region is executed by the implicit device thread and the encountering thread/task [on the host] waits at the construct until the execution of the region completes.

- If a target device is not present, or not supported, or not available, the target region is executed by the host device.

- If a construct creates a *data environment*, the data environment is created at the time the construct is encountered.
When an OpenMP program begins, each device has an initial *device data environment*.

Directives accepting data-mapping attribute clauses determine how an *original* variable is mapped to a *corresponding* variable in a device data environment:

- original: the variable on the host
- corresponding: the variable on the device
- the corresponding variable in the device data environment may share storage with the original variable (danger of data races)

If a corresponding variable is present in the enclosing device data environment, the new device data environment inherits the corresponding variable from the enclosing device.
Data environment is lexically scoped
- Data environment is destroyed at closing curly brace
- Allocated buffers/data are automatically released

Use target construct to
- Transfer control from the host to the device
- Establish a data environment (if not yet done)
- Host thread waits until offloaded region completed

#pragma omp target
 alloc(...)
 from(....)
```c
int main(int argc, const char* argv[]) {
    int n = 10240; float a = 2.0f; float b = 3.0f;
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Initialize x, y
    // Run SAXPY TWICE, assume modification of y in between
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
    for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
    }
    free(x); free(y); return 0;
}
```
```c
int main(int argc, const char* argv[]) {
    int n = 10240; float a = 2.0f; float b = 3.0f;
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Initialize x, y

    // Run SAXPY TWICE, assume modification of y in between
    #pragma acc data copyin(x[0:n])
    {

        #pragma acc parallel copy(y[0:n])
        #pragma acc loop
        for (int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }
        #pragma acc parallel copy(y[0:n])
        #pragma acc loop
        for (int i = 0; i < n; ++i){
            y[i] = b*x[i] + y[i];
        }
    }
    free(x); free(y); return 0;
}
```
int main(int argc, const char* argv[]) {
    int n = 10240; float a = 2.0f; float b = 3.0f;
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Initialize x, y

    // Run SAXPY TWICE, assume modification of y in between
    #pragma omp target data map(to:x[0:n])
    {
        #pragma omp target map(tofrom:y[0:n])
        #pragma omp parallel for
        for (int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }

        #pragma omp target map(tofrom:y[0:n])
        #pragma omp parallel for
        for (int i = 0; i < n; ++i){
            y[i] = b*x[i] + y[i];
        }
    }
    free(x); free(y); return 0;
}
int main(int argc, const char* argv[]) {
    int n = 10240; float a = 2.0f; float b = 3.0f;
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Initialize x, y

    // Run SAXPY TWICE, assume modification of y in between
    #pragma omp target data map(to:x[0:n])
    {
        #pragma omp target map( tofrom: y[0:n])
        #pragma omp teams
        #pragma omp distribute
        #pragma omp parallel for
        for (int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }
    }
    #pragma omp target map( tofrom: y[0:n])
    #pragma omp teams
    #pragma omp distribute
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
    }
}
free(x); free(y); return 0;
int main(int argc, const char* argv[]) {
    int n = 10240; float a = 2.0f; float b = 3.0f;
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Initialize x, y

    // Run SAXPY TWICE, assume modification of y in between
    #pragma omp target data map(to:x[0:n])
    {
        #pragma omp target map(tofrom:y[0:n])
        #pragma omp teams
        #pragma omp distribute
        #pragma omp parallel for
        for (int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }
    }
    #pragma omp target map(tofrom:y[0:n])
    #pragma omp teams
    #pragma omp distribute
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
    }
}
free(x); free(y); return 0;

Also possible (and equivalent) in OpenMP 4.0:

Combined directive

#pragma omp teams distribute parallel for
More related constructs

- **target update construct**
  - Makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses

- **declare target directive**
  - Specifies that [static] variables, functions (C, C++ and Fortran) and subroutines (Fortran) are mapped to (available on) a device

- **Support for Array Sections in C/C++ and Fortran, i.e. in map**
  - map just a slice of an array, not the whole array
  - describe the shape of a pointer (to some extent)
For asynchronous execution use the task construct and task dependencies:

```c
#pragma omp target data map(alloc:Z) 
{
    #pragma omp parallel for
    for (c = 0; c < nchunks; c += chunksz) 
    {
        #pragma omp task depend(out:c)
        #pragma omp target update map(to: Z[c:chunksz])

        #pragma omp task depend(in:c)
        #pragma omp target
        #pragma omp parallel for
        for (i = c; i < c + chunksz; i++)
            Z[i] = f(Z[i]);
    }
}
```

Finally, this example underlines the motivation for integrating accelerator support with the rest of OpenMP…
Vectorization
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
  - Auto-vectorization (works only if the code pattern is recognized by the compiler)

You need to trust your compiler to do the “right” thing.
Vectorize a loop nest

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

Syntax (C/C++)

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

Syntax (Fortran)

```fortran
!$omp simd [clause[[], clause],...]
do-loops
```
In this code one has to help the compiler enable vectorization:

```c
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 this 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
  \[ x_i = x_{\text{orig}} + i \times \text{linear-step} \]

- `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
SIMD Function Vectorization

- Declare one or more functions to be compiled for the target device

- Syntax (C/C++):

  ```
  #pragma omp declare simd [clause[[[,] clause],...]]
  [#pragma omp declare simd [clause[[[,] clause],...]]]
  [...]
  
  function-declarations-or-definitions
  ```

- Syntax (Fortran):

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

```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]);
    }
}
```

Parallelization + Vectorization
**SIMD Function 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

- As before: linear, aligned, reduction
Tasking
The **task dependence** is fulfilled when the predecessor task has completed

- **in** dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** clause.

- **out** and **inout** dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **in**, **out**, or **inout** clause.

- The list items in a **depend** clause may include array sections (linear algebra: slice of an array / matrix).
Concurrent Execution w/ Dep.

- Note: variables in the `depend` clause do not necessarily have to indicate the data flow, but they can

```c
void process_in_parallel() {
    #pragma omp parallel
    #pragma omp single
    {
        int x = 1;
        ...
        for (int i = 0; i < T; ++i) {
            #pragma omp task shared(x, ...) depend(out: x) // T1
            preprocess_some_data(...);
            #pragma omp task shared(x, ...) depend(in: x) // T2
            do_something_with_data(...);
            #pragma omp task shared(x, ...) depend(in: x) // T3
            do_something_independent_with_data(...);
        }
    } // end omp single, omp parallel
```

T1 has to be completed before T2 and T3 can be executed.

T2 and T3 can be executed in parallel.
The taskgroup Construct

- Specifies a wait on completion of child tasks and their descendent tasks
  - „deeper“ synchronization than `taskwait`, but
  - with the option to restrict to a subset of all tasks (as opposed to a `barrier`

C/C++

```c
#pragma omp taskgroup
... structured block ...
```

Fortran

```fortran
!$omp taskgroup
... structured block ...
!$omp end task
```

RZ: Christian Terboven
Cancellation of OpenMP Tasks

- Cancellation only acts on tasks of group by taskgroup
  - The encountering tasks jumps to the end of its task region
  - Any executing task will run to completion (or until they reach a cancellation point region)
  - Any task that has not yet begun execution may be discarded (and is considered completed)

- Tasks cancellation also occurs, if a parallel region is canceled.
  - But not if cancellation affects a worksharing construct.
binary_tree_t* search_tree_parallel(binary_tree_t* tree, int value) {
    binary_tree_t* found = NULL;
    #pragma omp parallel shared(found,tree,value)
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                found = search_tree(tree, value);
            }
        }
    }
    return found;
}
Task Cancellation Example

```c
binary_tree_t* search_tree(
    binary_tree_t* tree, int
value,
    int level) {
    binary_tree_t* found = NULL;
    if (tree) {
        if (tree->value == value) {
            found = tree;
        } else {
            #pragma omp task shared(found)
            {
                binary_tree_t* found_left;
                found_left =
                search_tree(tree->left,
value);
                if (found_left) {
                    #pragma omp atomic write
                    found = found_left;
                }
            }
            #pragma omp taskwait
            }
        } else {
            #pragma omp task shared(found)
            {
                binary_tree_t* found_right;
                found_right =
                search_tree(tree->right,
value);
                if (found_right) {
                    #pragma omp atomic write
                    found = found_right;
                }
            }
            #pragma omp cancel taskgroup
            }
        }
    return found;
```
Reductions
Example: Bounding Box Code

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

```cpp
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
OpenMP 3.0 introduced Worksharing support for iterator loops

#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

#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++) {
        ...

Bounding Box w/ OpenMP 4.0
Feature List
Overview of major 4.0 additions

- **Presented today**
  - Device constructs
  - Task dependences and task groups
  - Support for array sections (including in C and C++)
  - SIMD constructs
  - Cancellation
  - User-defined reductions

- **Not covered today**
  - Thread affinity control  (see backup slides if you are interested)
  - Sequentially consistent atomics
  - Initial support for Fortran 2003
  - Display of initial OpenMP internal control variables
Thank you for your attention.

Past OpenMP 4.0 Events, material available publically / for audience

- Hartree Centre Summer School, Daresbury, UK: 14.-19.07.2013
- Parallel Programming Course PPCES, Aachen: 29.07.-02.08.2013
- Advanced OpenMP Tutorial @ Euro-Par: 27.08.2013
- aiXcelerate Workshop, Aachen: 07.-10.10.2013
- Rolf Rabenseifner covers selected new features in his OpenMP courses

Future OpenMP 4.0 Events

- Advanced OpenMP Tutorial @ SC13: 18.11.2013 (full day)
- Several talks and presentations at the OpenMP Booth @ SC13: 18.-21.11.2013

heise Developer: “Die wichtigsten Neuerungen von OpenMP 4.0“

Quick Syntax Reference soon to be available from www.openmp.org
Backup
OpenMP 4.0: Places + Policies (1)

- Define OpenMP Places
  - set of OpenMP 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

- Define a set of OpenMP Thread Affinity Policies
  - SPREAD: spread OpenMP threads evenly among the places
  - CLOSE: pack OpenMP threads near master thread
  - MASTER: collocate OpenMP thread with master thread

- Goals
  - user has a way to specify where to execute OpenMP threads
  - locality between OpenMP threads / less false sharing / memory bandwidth
Example’s Objective:
- separate cores for outer loop and near cores for inner loop

Outer Parallel Region: proc_bind(spread)
Inner Parallel Region: proc_bind(close)

- spread creates partition, compact binds threads within respective partition

OMP_PLACES={0,1,2,3}, {4,5,6,7}, ... = {0-4}:4:8

#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)

Example
- initial
- spread 4
- close 4
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: