OpenMP 5.0 for Accelerators and What Comes Next

Tom Scogland and Bronis de Supinski

LLNL

LLNL-PRES-767542

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

OpenMP 5.0 was ratified in November

- Addressed several major open issues for OpenMP
- Did not break (most?) existing code
  - → One possible issue: `nonmonotonic` default
- Includes 293 passed tickets: *lots* of new changes

Major new features in OpenMP 5.0

- Significant extensions to improve usability and offload flexibility
  - → OpenMP contexts, `metadirective` and `declare variant`
  - → Addition of `requires` directive, including support for unified shared memory
  - → Memory allocators and support for deep memory hierarchies
  - → Descriptive `loop` construct
  - → Release/acquire semantics added to memory model
- Host extensions that sometimes help
  - → Ability to quiesce OpenMP threads
  - → Support to print/inspect affinity state
  - → Support for C/C++ array shaping
- First (OMPT) and third (OMPD) party tool support

Major new features in OpenMP 5.0

- Some significant extensions to existing functionality
  - → Verbosity reducing changes such as implicit `declare target` directives
  - → User defined mappers provide deep copy support for map clauses
  - → Support for reverse offload
  - → Support for task reductions , including on `taskloop` construct, task affinity, new dependence types, depend objects and detachable tasks
  - → Allows `teams` construct outside of `target` (i.e., on host)
  - → Supports collapse of non-rectangular loops
  - → Scan extension of reductions
- Major advances for base language normative references
  - → Completed support for Fortran 2003
  - → Added Fortran 2008, C11, C++11, C++14 and C++17

Clarifications and minor enhancements

- Supports collapse of imperfectly nested loops
- Supports `!=` on C/C++ loops, and range for `for(auto &x:range)`
- Adds `conditional` modifier to `lastprivate`
- Support use of any C/C++ *lvalue* in `depend` clauses
- Permits `declare target` on C++ classes with virtual members
- Clarification of `declare target` C++ initializations
- Adds `task` modifier on many `reduction` clauses
- Adds `depend` clause to `taskwait` construct

An OpenMP 4 example

```
typedef struct mypoints {
    int len;
    double *needed_data;
    double useless_data[500000];
} mypoints_t;
```

- Heterogeneous programming requires map clauses to transfer (ownership of) data to target devices

- map can't provide deep copy on a single construct

- No support for unified memory in portable code

```
#pragma omp declare target
int do_something_with_p(mypoints_t &p_ref);
#pragma omp end declare target

mypoints_t * p = new_array_of_mypoints_t(N);

#pragma omp target enter data map(p[0:N])
for(int i=0; i<N; ++i){
  #pragma omp target enter data \
              map(p[i].needed_data[0:p[i].len])
}

#pragma omp target // can't express map here
{
    do_something_with_p(*p);
}
```

The `requires` Construct

```
typedef struct mypoints {
    int len;
    double *needed_data;
    double useless_data[500000];
} mypoints_t;
```

- Informs the compiler that the code **requires** an optional feature or setting to work

- OpenMP 5.0 adds the requires construct so that a program can declare that it assumes shared memory between devices

```
#pragma omp declare target
int do_something_with_p(mypoints_t &p_ref);
#pragma omp end declare target

#pragma omp requires unified_shared_memory

mypoints_t * p = new_array_of_mypoints_t(N);

#pragma omp target // no map clauses needed
{
    do_something_with_p(*p);
}
```

Implicit `declare target`

```
typedef struct mypoints {
    int len;
    double *needed_data;
    double useless_data[500000];
} mypoints_t;
```

- Heterogeneous programming requires compiler to generate versions of functions for the devices on which they will execute
- Generally requires the programmer to inform compiler of the devices on which the functions will execute
- OpenMP 5.0 requires the compiler to assume device versions exist and to generate them when it can "see" the definition and a use on the device

```
// no declare target needed
int do_something_with_p(mypoints_t &p_ref);

#pragma omp requires unified_shared_memory

mypoints_t * p = new_array_of_mypoints_t(N);

#pragma omp target // no map clauses needed
{
    do_something_with_p(*p);
}
```

Deep Copy with declare mapper

```
typedef struct mypoints {
    int len;
    double *needed_data;
    double useless_data[500000];
} mypoints_t;
```

- Not all devices support shared memory so requiring it makes a program less portable
- Painstaking care was required to map complex data before 5.0
- OpenMP 5.0 adds deep copy support so that programmer can ensure that compiler correctly maps complex (pointer-based) data

```
// no declare target needed
int do_something_with_p(mypoints_t *p);

#pragma omp declare mapper(mypoints_t v)\
        map(v.len, v.needed_data,       \
            v.needed_data[0:v.len])

mypoints_t * p = new_array_of_mypoints_t(N);

#pragma omp target map(p[:N])
{
    do_something_with_p(p);
}
```

Reverse Offload

- Why only offload from host to device?
- Why pessimize every launch when you only sometimes need to go back to the host?

Reverse Offload

```
#pragma omp requires reverse_offload

#pragma omp target map(inout: data[0:N])
{
  do_something_offloaded(data);
  #pragma omp target device(ancestor: 1)
  printf("back on the host right now\n");
  do_something_after_print_completes();
  #pragma omp target device(ancestor: 1)\
                  map(inout: data[0:N])
  MPI_Isend(... data ...);
  do_more_work_after_MPI();
}
```

Reverse Offload: take care!

```
#pragma omp requires reverse_offload

#pragma omp target teams parallel num_teams(T) num_threads(N)
{
  #pragma omp target device(ancestor: 1)
  printf("back on the host right now\n");
  // called N*T times on the host, probably serially!
}
```

Execution Contexts

- Context describes lexical “scope” of an OpenMP construct and it’s lexical nesting in other OpenMP constructs:

```
// context = {}
#pragma omp target teams
{
  // context = {target, teams}
  #pragma omp parallel
  {
    // context = {target, teams, parallel}
    #pragma omp simd aligned(a:64)
    for (...)
    {
      // context = {target, teams, parallel, simd(aligned(a:64), simdlen(8), notinbranch) }
      foo(a);
}}}
```

- Contexts also apply to `metadirective`

Meta directive

The directive directive

- Started life many years, at least 5, ago as the super_if
- Especially important now that we have target constructs
- A metadirective is a directive that can specify multiple directive variants of which one may be conditionally selected to replace the metadirective based on the enclosing OpenMP context.

```
#pragma omp metadirective \
            when(device={kind(gpu)}: parallel for)\
            default( target teams distribute parallel for )
for (i= lb; i< ub; i++)
   v3[i] = v1[i] * v2[i];
   ...
```

Meta directive

The directive directive

- Started life many years, at least 5, ago as the super_if
- Especially important now that we have target constructs
- A metadirective is a directive that can specify multiple directive variants of which one may be conditionally selected to replace the metadirective based on the enclosing OpenMP context.

```
#pragma omp target teams distribute
for (i= lb; i< ub; i++)
   v3[i] = v1[i] * v2[i];
   ...
```

When compiling to be called on a gpu

Meta directive

The directive directive

- Started life many years, at least 5, ago as the super_if
- Especially important now that we have target constructs
- A metadirective is a directive that can specify multiple directive variants of which one may be conditionally selected to replace the metadirective based on the enclosing OpenMP context.

When compiling for a anything that is not a gpu!

```
#pragma omp target teams distribute parallel for
for (i= lb; i< ub; i++)
   v3[i] = v1[i] * v2[i];
   ...
```

Meta directive

The directive directive

- Started life many years, at least 5, ago as the super_if
- Especially important now that we have target constructs
- A metadirective is a directive that can specify multiple directive variants of which one may be conditionally selected to replace the metadirective based on the enclosing OpenMP context.

When compiling for both

```
#pragma omp target teams distribute parallel for
for (i= lb; i< ub; i++)
   v3[i] = v1[i] * v2[i];
   ...
```

```
#pragma omp parallel for
for (i= lb; i< ub; i++)
   v3[i] = v1[i] * v2[i];
   ...
```

Declare variant directive

- The declare variant directive declares a specialized variant of a base function and specifies the context in which that specialized variant is used. The declare variant directive is a declarative directive.
- Combines proposed extensions for DECLARE SIMD and DECLARE TARGET into one that works anywhere.
- Reuse context selector mechanism used by meta directive

Declare variant directive

```
#pragma omp declare variant( int important_stuff(int x) ) \
                    match( context={target,simd} device={arch(nvptx)} )
int important_stuff_nvidia(int x){
      /* Specialized code for NVIDIA target */
}

#pragma omp declare variant( int important_stuff(int x) ) \
                    match( context={target, simd(simdlen(4))}, device={isa(avx2)}
)
 __m256i _mm256_epi32_important_stuff(__m256i x) {
     /* Specialized code for simdloop called on an AVX2 processor */
}
...
int y =important_stuff(x);
```

Declare variant directive

This may not be the supported name!

```
#pragma omp declare variant( int important_stuff(int x) ) \
                    match( context={target,simd} device={arch(nvptx)} )
int important_stuff_nvidia(int x){
      /* Specialized code for NVIDIA target */
}

#pragma omp declare variant( int important_stuff(int x) ) \
                    match( context={target, simd(simdlen(4))}, device={isa(avx2)}
)
 __m256i _mm256_epi32_important_stuff(__m256i x) {
     /* Specialized code for simdloop called on an AVX2 processor */
}
...
int y =important_stuff(x);
```

When compiling for NVIDA GPUS the compiler translates this to important_stuff_nvidia(x);

Declare variant directive

```
#pragma omp declare variant( int important_stuff(int x) ) \
                    match( context={target,simd} device={arch(nvptx)} )
int important_stuff_nvidia(int x){
      /* Specialized code for NVIDIA target */
}

#pragma omp declare variant( int important_stuff(int x) ) \
                    match( context={target, simd(simdlen(4))}, device={isa(avx2)}
)
 __m256i _mm256_epi32_important_stuff(__m256i x) {
     /* Specialized code for simdloop called on an AVX2 processor */
}
...
int y =important_stuff(x);
```

When compiling for AVX2 the compiler translates this to __m256i _mm256_epi32_important_stuff (x);

#pragma omp loop

- Introduced as `#pragma omp concurrent` in TR6
  - → A loop construct specifies that the iterations of the associated loops may execute concurrently and permits the encountering thread(s) to execute the loop accordingly.
- Why?
  - → It's descriptive!
  - → Enables the compiler to make certain complex optimizations that would require dependency analysis
- Limitations
  - → Not a complete replacement for do/for, yet!
  - → User responsible for bindings, teams, parallel, thread, of orphaned constructs.

OMP loop example

```
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
    #pragma omp loop
    for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
    }
  }
}
```

Generate Parallelism

Assert to the compiler that it is safe to parallelize the next loop

loop: Reprising an OpenACC Example

"Why can't OpenMP do this?"

```
while (error > tol && iter < iter_max) {
  error = 0.0;
#pragma omp parallel for reduction(max : error)
  for (int j = 1; j < n - 1; j++) {
#pragma omp simd
    for (int i = 1; i < m - 1; i++) {
      Anew[j][i] = 0.25*(A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i]);
      error = fmax(error, fabs(Anew[j][i] - A[j][i]));
    }
  }
#pragma omp parallel for
  for (int j = 1; j < n - 1; j++) {
#pragma omp simd
    for (int i = 1; i < m - 1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
  if (iter++ % 100 == 0)
    printf("%5d, %0.6f\n", iter, error);
}
```

```
while ( error > tol && iter < iter_max )
{
  error = 0.0;
#pragma acc parallel loop reduction(max:error)
  for ( int j = 1; j < n - 1; j++)
  {
#pragma acc loop reduction(max:error)
    for ( int i = 1; i < m - 1; i++ )
    {
      Anew[j][i] = 0.25 * ( A[j][i + 1] + A[j][i - 1]
                          + A[j - 1][i] + A[j + 1][i]);
      error = fmax( error, fabs(Anew[j][i] - A[j][i]));
    }
  }
#pragma acc parallel loop
  for ( int j = 1; j < n - 1; j++)
  {
#pragma acc loop
    for ( int i = 1; i < m - 1; i++ )
    {
      A[j][i] = Anew[j][i];
    }
  }
  if (iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

loop: Reprising an OpenACC Example

It can!

```
while (error > tol && iter < iter_max) {
  error = 0.0;
#pragma omp target teams
#pragma omp loop reduction(max : error)
  for (int j = 1; j < n - 1; j++) {
#pragma omp loop reduction(max : error)
    for (int i = 1; i < m - 1; i++) {
      Anew[j][i] = 0.25*(A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i]);
      error = fmax(error, fabs(Anew[j][i] - A[j][i]));
    }
  }
#pragma omp target teams
#pragma omp loop
  for (int j = 1; j < n - 1; j++) {
#pragma omp loop
    for (int i = 1; i < m - 1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
  if (iter++ % 100 == 0)
    printf("%5d, %0.6f\n", iter, error);
}
```

```
while ( error > tol && iter < iter_max )
{
  error = 0.0;
#pragma acc parallel loop reduction(max:error)
  for ( int j = 1; j < n - 1; j++)
  {
#pragma acc loop reduction(max:error)
    for ( int i = 1; i < m - 1; i++ )
    {
      Anew[j][i] = 0.25 * ( A[j][i + 1] + A[j][i - 1]
                          + A[j - 1][i] + A[j + 1][i]);
      error = fmax( error, fabs(Anew[j][i] - A[j][i]));
    }
  }
#pragma acc parallel loop
  for ( int j = 1; j < n - 1; j++)
  {
#pragma acc loop
    for ( int i = 1; i < m - 1; i++ )
    {
      A[j][i] = Anew[j][i];
    }
  }
  if (iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Allocators

- New clause on all constructs with data sharing clauses:
  - `allocate( [allocator:] list )`
- Allocation:
  - `omp_alloc(size_t size, omp_allocator_t *allocator)`
- Deallocation:
  - `omp_free(void *ptr, const omp_allocator_t *allocator)`
  - allocator argument is optional
- `allocate` directive
  - Standalone directive for allocation, or declaration of allocation stmt.

Allocators

```
void allocator_example(omp_allocator_t *my_allocator) {
    int a[M], b[N], c;
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(b) // controlled by OMP_ALLOCATOR
                            // and/or omp_set_default_allocator
    double *p = (double *) omp_alloc(N*M*sizeof(*p), my_allocator);
    #pragma omp parallel private(a) allocate(my_allocator:a)
    {
        some_parallel_code();
    }

    #pragma omp target firstprivate(c) allocate(omp_const_mem_alloc:c)
    {
         #pragma omp parallel private(a) allocate(omp_high_bw_mem_alloc:a)
         some_other_parallel_code();
    }

    omp_free(p);
}
```

Allocators: team-local memory

```
// CUDA
__global__ void staticReverse(int *d, int n) {
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
// OpenMP 5
#pragma omp target parallel private(s)
{ int s[64];
  #pragma omp allocate(s) allocator(omp_pteam_mem_alloc)
  int t = omp_get_thread_num();
  int tr = n-t-1;
  s[t] = d[t];
  #pragma omp barrier
  d[t] = s[tr];
}
```

Native Prefix Scan Support

```
int x = 0;
#pragma omp parallel for simd\
        reduction(inscan,+: x)
for (i = 0; i < n; ++i) {
  x += A[i];
  #pragma omp scan inclusive(x) //exclusive(x)
  B[i] = x;
}
```

A	1	2	3	4	5	6	...
B (inclusive)	1	3	6	10	15	21	...
B (exclusive)	0	1	3	6	10	15	...

OpenMP 5.1 will be released in November 2020

- Proceedings of the IEEE article on vision: “The Ongoing Evolution of OpenMP”
  - → Broadly support on-node performant, portable parallelism
  - → OpenMP 5.0 fits within that vision
  - → OpenMP 5.1 will refine how OpenMP 5.0 realizes it
  - → OpenMP 6.0 will be a major step to further realizing it
- Expect issues from detailed implementation and use of OpenMP 5.0, which is big and will require time to implement
- Guarantee OpenMP 5.1 will not break existing code
- Clarifications, corrections and maybe some small extensions
  - → Improved native device support (e.g., CUDA streams)
  - → May add `taskloop` dependences
  - → Other small extensions, must entail small implementation burden

OpenMP 6.0 will be released in November 2023

- Deeper support for descriptive and prescriptive control
- More support for memory affinity and complex hierarchies
- Support for pipelining, other computation/data associations
- Continued improvements to device support
  - → Extensions of deep copy support (serialize/deserialize f'ns)
- Task-only, unshackled or free-agent threads
- Re-usable tasks or task graphs
- Event-driven parallelism
- Completing support for new normative references
- 38 5.1 tickets already; 2 tickets already deferred to 6.0

Visit www.openmp.org for more information