AURORA ESP TRAINING WEBINAR

Argonne NATIONAL LABORATORY

OPENMP 5.0

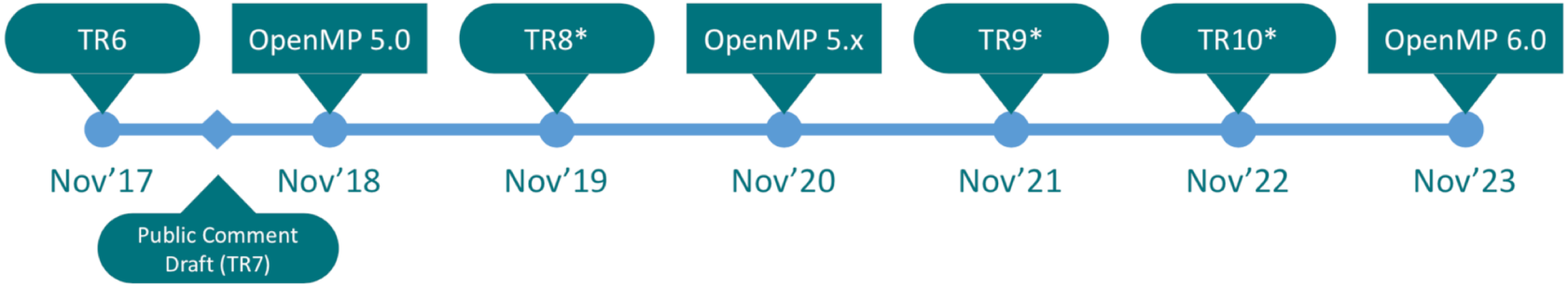
JAEHYUK KWACK
ALCF Perf. Engr. group

This material is mostly based on slides of Michael Klemm (OpenMP.org) and Tom Scogland (LLNL).

June 18th, 2019

OPENMP ROADMAP

- OpenMP 4.5 Specifications were released in November 2015. (359 pages)
- OpenMP 5.0 Specifications were released in November 2018. (646 pages)
- OpenMP 5.x Specifications will be released in November 2020. (??? pages)



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

AN OPENMP 4.5 EXAMPLE

- map clause is required to transfer data to target devices
- map cannot provide deep copy on a single construct
- No support for unified memory

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

```
#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);
}
```

“REQUIRES” CONSTRUCT

- 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

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

```
#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”

- 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

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

```
// 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"

- 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

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

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

#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

When you need to go back to the host in a target region

```
#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();
}
```

```
#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!
}
```

METADIRECTIVE

- 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 map(to:v1,v2) map(from:v3)
#pragma omp metadirective \
        when( device={arch(nvptx)}: teams loop ) \
        default( parallel loop )
for (i = lb; i < ub; i++)
        v3[i] = v1[i] * v2[i];
```

```
!$omp begin metadirective &
        when( implementation={unified_shared_memory}: target ) &
        default( target map(mapper(vec_map),tofrom: vec) )
!$omp teams distribute simd
do i=1, vec%size()
        call vec(i)%work()
end do
!$omp end teams distribute simd
!$omp end metadirective
```

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.

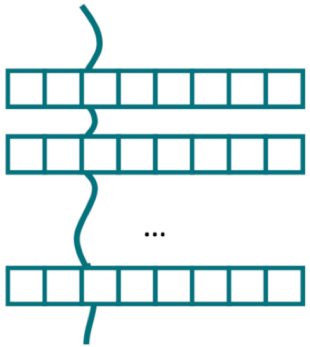
```
#pragma omp declare variant( int important_stuff(int x) ) \
             match( context={target,simd} device={arch(nvptx)} )
int important_stuff_nvidia(int x)
{ /* Specialized code for NVIDIA target */ }
#pragma omp declare variant(int important_stuff (int x)) \
             match( context={target,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


...
int y =important_stuff(x);
```

LOOP DIRECTIVE

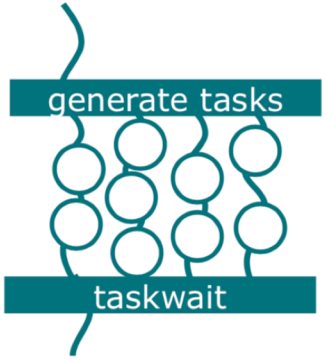
- Existing loop constructs are tightly bound to execution model:

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


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




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



- The loop construct is meant to let the OpenMP implementation pick the right parallelization scheme for a parallel loop.

LOOP CONSTRUCT

Simplified loop construct on OpenMP 5

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

OpenMP 4.5 Multi-level Device Parallelism

Simplifying Multi-level Device Parallelism

LOOP: REPRISING AN OPENACC EXAMPLE

```
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);
}
```

OpenMP 4.5

```
while (error > tol && iter < iter_max) {
  error = 0.0;
  #pragma omp target teams distribute reduction(max:error)
  for (int j = 1; j < n - 1; j++) {
    #pragma omp parallel for 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 distribute
  for (int j = 1; j < n - 1; j++) {
    #pragma omp parallel for
    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

```
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);
}
```

OpenMP 5.0

```
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);
}
```

MEMORY 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.

EXAMPLE: MEMORY 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) // on target; must be compile-time expr
	{
		#pragma omp parallel private(a) allocate(omp_high_bw_mem_alloc:a)
		{
			some_other_parallel_code();
		}
	}
	omp_free(p);
}
```

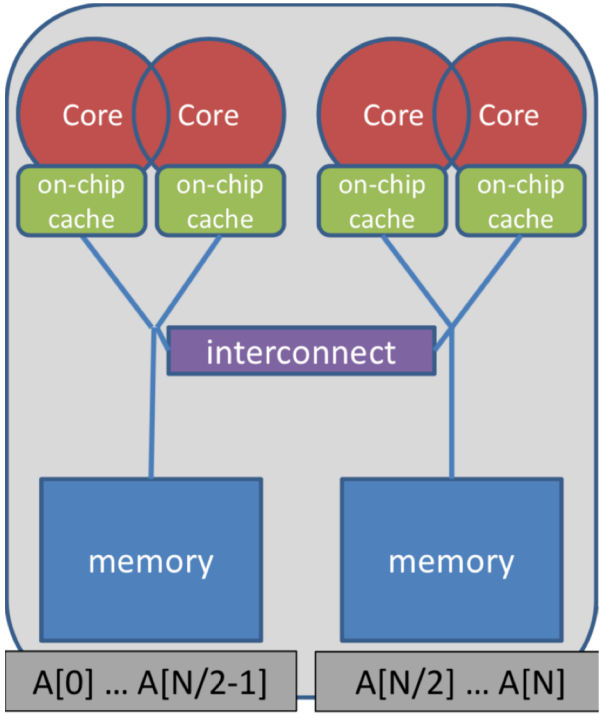
EXAMPLE: TEAM-LOCAL MEMORY

```
// 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];
}
```

PARTITIONING MEMORY

Optimized memory access by partitioning on OpenMP 5

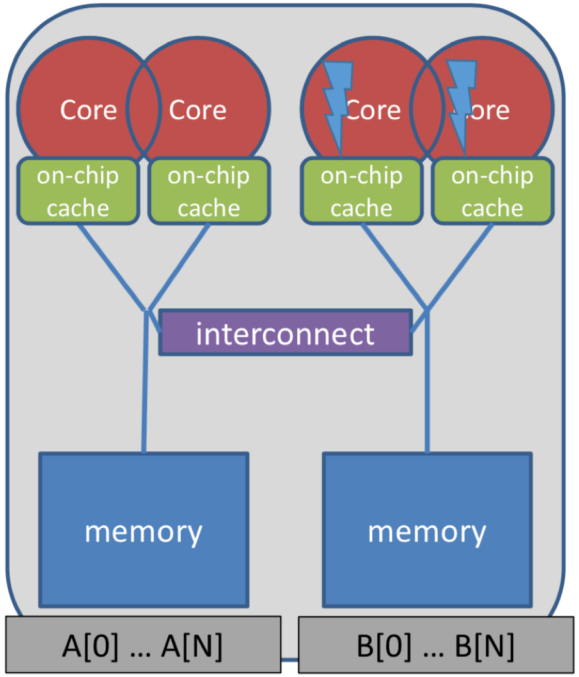
```
void allocator_example() {
        double *array;

        omp_allocator_t *allocator;
        omp_alloctrait_t traits[] = {
                {OMP_ATK_PARTITION, OMP_ATV_BLOCKED}
        };
        int ntraits = sizeof(traits) / sizeof(*traits);
        allocator = omp_init_allocator(omp_default_mem_space, ntraits, traits);

        array = omp_alloc(sizeof(*array) * N, allocator);
        #pragma omp parallel for proc_bind(spread)
        for (int i = 0; i < N; ++i) {
                important_computation(&array[i]);
        }

        omp_free(array);
}
```



TASK-TO-DATA AFFINITY

OpenMP 5 supports affinity hints for OpenMP tasks

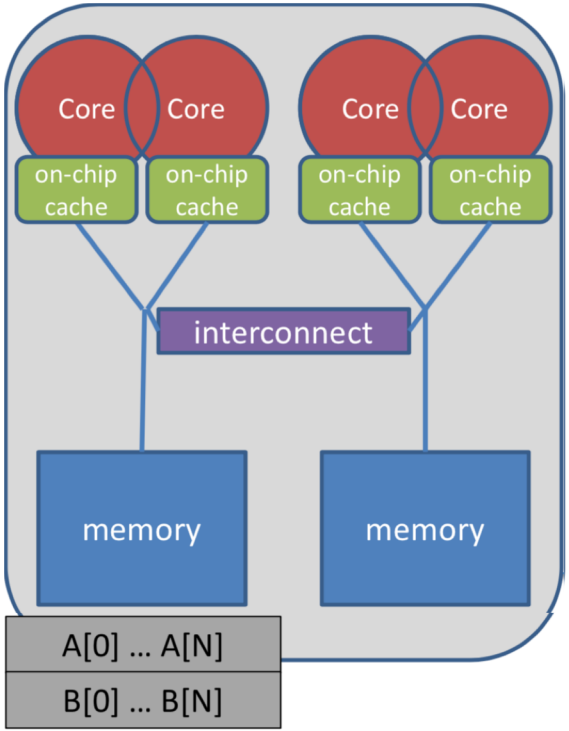
```
void task_affinity() {
      double* B;
      #pragma omp task shared(B)
      {
            B = init_B_and_important_computation(A);
      }
      #pragma omp task firstprivate(B)
      {
            important_computation_too(B);
      }
      #pragma omp taskwait
}
```



TASK-TO-DATA AFFINITY

OpenMP 5 supports affinity hints for OpenMP tasks

```
void task_affinity() {
      double* B;
      #pragma omp task shared(B) affinity(A[0:N])
      {
            B = init_B_and_important_computation(A);
      }
      #pragma omp task firstprivate(B) affinity(B[0:N])
      {
            important_computation_too(B);
      }
      #pragma omp taskwait
}
```



TASK REDUCTIONS

- Task reductions extend traditional reductions to arbitrary task graphs
- Extend the existing task and taskgroup constructs
- Also work with the taskloop construct

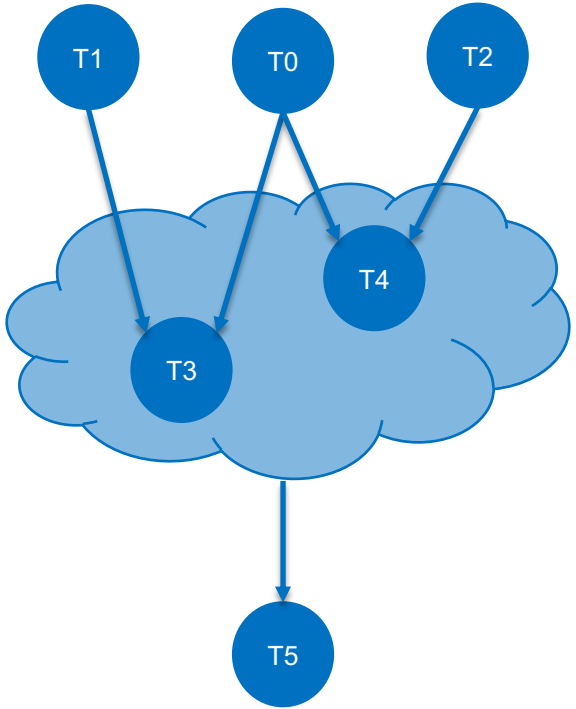
```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup task_reduction(+: res)
    {
      while (node) {
        #pragma omp task in_reduction(+: res) \
                         firstprivate(node)
        {
          res += node->value;
        }
        node = node->next;
      }
    }
  }
}
```

`depend(mutexinoutset:var)`
only one task using *var* can run at a time

NEW TASK DEPENDENCIES

OpenMP 5 provides new task dependencies for better performance

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res) //T0
  res = 0;
  #pragma omp task depend(out: x) //T1
  long_computation(x);
  #pragma omp task depend(out: y) //T2
  short_computation(y);
  #pragma omp task depend(in: x) depend(mutexinoutset: res) //T3 – OpenMP 5
  res += x;
  #pragma omp task depend(in: y) depend(mutexinoutset: res) //T4 – OpenMP 5
  res += y;
  #pragma omp task depend(in: res) //T5
  std::cout << res << std::endl;
}
```



ASYNCHRONOUS API INTERACTION

- Some APIs are based on asynchronous operations
  - MPI asynchronous send and receive
  - Asynchronous I/O
  - CUDA and OpenCL stream-based offloading
  - In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: CUDA memory transfers

```
do_something();
cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
do_something_else();
cudaStreamSynchronize(stream);
do_other_important_stuff(dst);
```

- Programmers need a mechanism to marry asynchronous APIs with the parallel task model of OpenMP
  - How to synchronize completions events with task execution?

ASYNCHRONOUS API INTERACTION

Use OpenMP Tasks

```
void cuda_example() {
        #pragma omp task // task A
        {
                do_something();
                cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
        }
        #pragma omp task // task B
        {
                do_something_else();
        }
        #pragma omp task // task C
        {
                cudaStreamSynchronize(stream);
                do_other_important_stuff(dst);
        }
}
```

Race condition between the tasks A & C, task C may start execution before task A enqueues memory transfer.

ASYNCHRONOUS API INTERACTION

Detaching Tasks

1. Task detaches
2. `taskwait` contruct cannot complete
3. Signal event for completion
4. Task completes and `taskwait` can continue

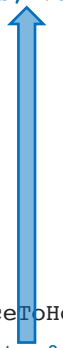
```
omp_event_t *event;
void detach_example() {
  #pragma omp task detach(event)
  {
    important_code();
  } 1
  #pragma omp taskwait 2 4
}
```

```
omp_fulfill_event(event); 3     // some other thread/task
```

ASYNCHRONOUS API INTERACTION

Detachable Tasks and Asynchronous APIs

```
void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *cb_dat) {
  3  omp_fulfill_event((omp_event_t *) cb_data);
}
void cuda_example() {
      omp_event_t *cuda_event;
      #pragma omp task detach(cuda_event)      // task A
      {
            do_something();
            cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
            cudaStreamAddCallback(stream, callback, cuda_event, 0);
      } 1
      #pragma omp task                         // task B
            do_something_else();
      #pragma omp taskwait 2 4
      #pragma omp task                         // task C
      {
            do_other_important_stuff(dst);
}     }
```



1. Task A detaches
2. `taskwait` does not continue
3. When memory transfer completes, callback is invoked to signal the event for task completion
4. `taskwait` continues, task C executes

ASYNCHRONOUS API INTERACTION

Detachable Tasks and Asynchronous APIs

```
void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *cb_dat) {
   2  omp_fulfill_event((omp_event_t *) cb_data);
}
void cuda_example() {
       omp_event_t *cuda_event;
       #pragma omp task depend(out:dst) detach(cuda_event)   // task A
       {
              do_something();
              cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
              cudaStreamAddCallback(stream, callback, cuda_event, 0);
       } 1
       #pragma omp task                          // task B
              do_something_else();
       #pragma omp task depend (in: dst)     3 // task C
       {
              do_other_important_stuff(dst);
}      }
```

1. Task A detaches and task C will not execute because of its unfulfilled dependency on A
2. The Memory transfer completes and the callback is invoked to signal the event for task completion
3. Task A completes and C's dependency is fulfilled

THANK YOU!