Evaluating Support for OpenMP Offload Features

UDEL: Jose Monsalve (josem@udel.edu), Sunita Chandrasekaran (schandra@udel.edu), Kyle Friedline (utimatu@udel.edu)

ORNL: Swaroop Pophale (pophaless@ornl.gov), Oscar Hernandez (oscar@ornl.gov), David Berndholt (bernholdtde@ornl.gov)

Abstract

OpenMP has evolved to meet the rapid development in hardware platforms including heterogenous programming. DOE applications tend to push the bleeding edge of features ratified in the OpenMP specification and tend to expose the rough edges of the features' implementations. The software harness on DOE supercomputers (e.g. Titan and Summit) include Cray, Clang, Flang, XL and GCC compilers which claim partial support for the latest features in OpenMP 4.0+. This work focuses on evaluating such support across compiler implementations, focusing on OpenMP 4.5 target offload directives. Our preliminary evaluation consist of a tests suite, as well as performance comparison.

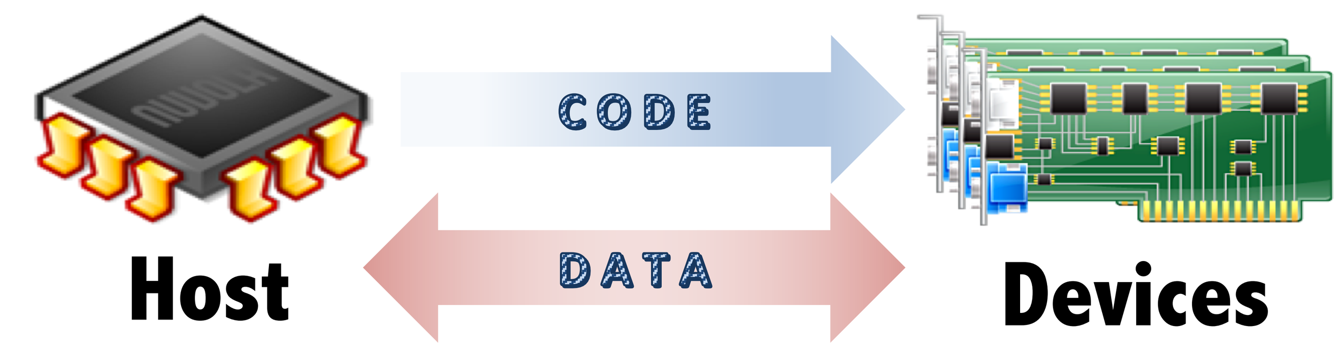
Our tests not only evaluate the OpenMP implementations but also expose ambiguities in the OpenMP 4.5 specification. We see this as a synergistic effort to help identify and correct features that are required by DOE applications and prevent deployment delays later on.

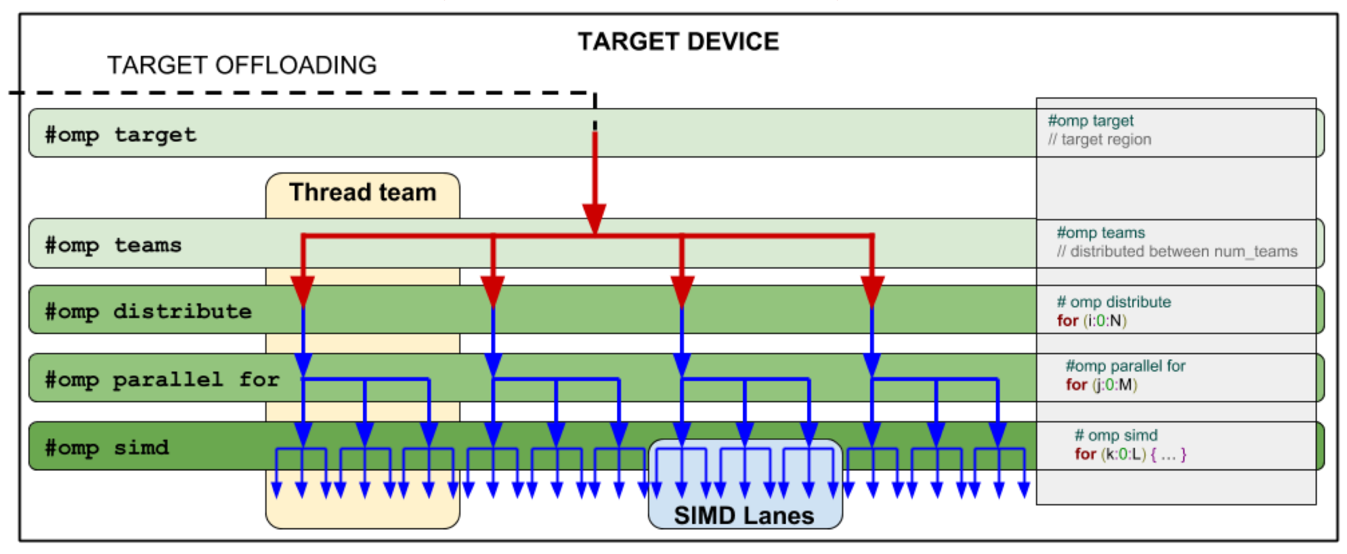
OpenMP 4.5 offloading

OpenMP abstract machine for offloading features is host centric: Offloading directives hint the compiler to create device executable regions of code, as well as code and data movement between host and device.

OpenMP frees the programmer from bookkeeping data allocation and movement, as well as separate compilation of code for host and device.

OpenMP 4.5 in particular provides more control to the programmer to handle data movement between host and device.





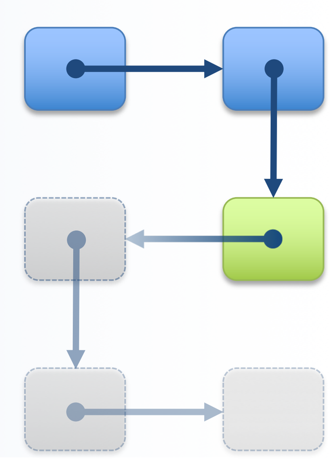
- Target region for device code generation
- Device-host data management
- Conditional execution of code in device
- Target device selection during runtime

```
#pragma omp target map(tofrom: myVar) if(myCondition) device(2)
{   myVar ++;   }
```

Complex Test Cases

```
1  typedef struct node {
2      double data;
3      struct node *next;
4  } node_t;
5
6  void map_ll(node_t *head) {
7      if (!head) return;
8  #pragma omp target enter data map(to:head[:1])
9      while(head->next) {
10         // Note: explicit attachment
11         node_t *cur = head->next;
12 #pragma omp target enter data map(to:cur[:1])
13 #pragma omp target
14         {
15             head->next = cur;
16         }
17     }
18 }
19
20 void unmap_ll(node_t *head) {
21     if (!head) return;
22 #pragma omp target exit data map(from:head[0].data)
23     while(head->next) {
24         // Note: only copies back the data element to avoid
           overwriting next pointer
25 #pragma omp target exit data map(from:head[0].next[0].data)
26     }
27 }
```

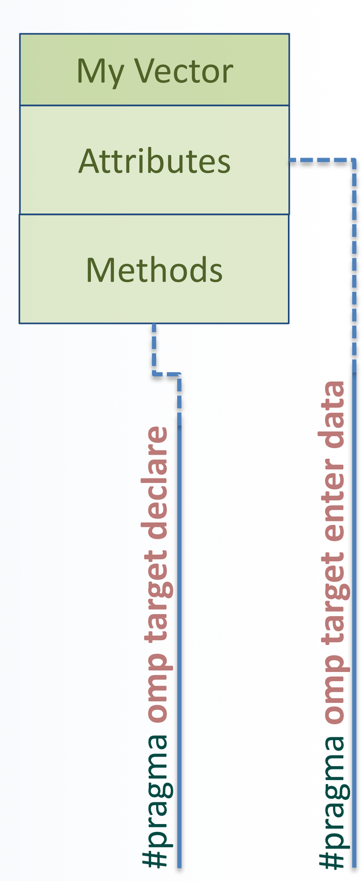
Mapping Linked list to device:
The *map_ll* function (line 6) uses *target enter data* directive to first map the head of the linked list, followed by mapping the pointer to the next node of the list and assigning it on the device. The *unmap_ll* (line 20) function explicitly copies the data using map(from:…) and *target exit data*.



```
1  #define RealType double
2  #define N 10
3
4  #pragma omp declare target
5  class MyVector
6  {
7  public:
8
9    inline RealType operator()(int i, int j, int k) const
10   {
11     return X[k*Length[2]+(j*Length[1]+i)];
12   }
13
14   inline RealType& operator()(int i, int j, int k)
15   {
16     return X[k*Length[2]+(j*Length[1]+i)];
17   }
18
19   MyVector(int l, int m, int n)
20   {
21     Length[0] = l;
22     Length[1] = m;
23     Length[2] = n;
24     X = new RealType[l*m*n];
25   }
26
27   RealType *& getData() { return X; }
28
29   RealType * getData() const { return X; }
30
31   int getSize() const { return Length[0]*Length[1]*Length[2]; }
32
33   int Length[3];
34   RealType * X;
35 };
36 #pragma omp end declare target
37
38 int main () {
39
40   MyVector gamma (N, N, N);
41   int size = gamma.getSize();
42
43 #pragma omp target enter data map(to:gamma)
44 #pragma omp target enter data map(to:gamma.X[0:size]) map(to:gamma.Length)
45
46 #pragma omp target
47   for(int i = 0 ; i < N ; i++)
48     for(int j = 0 ; j < N ; j++)
49       for(int k = 0 ; k < N ; k++)
50         gamma(i,j,k) = 1.0;
51
52 #pragma omp target exit data map(from:gamma.X[0:size])
53
54   for(int i = 0 ; i < N ; i++)
55     for(int j = 0 ; j < N ; j++)
56       for(int k = 0 ; k < N ; k++)
57         cout << gamma(i,j,k) << " ";
58   return 0;
59 }
```

Deep Copy of Classes:
This code came from analyzing a full scale ECP application. It uses the *declare target* directive (line 4 to 36) to ensures that procedures and global variables can be executed and data can be accessed on the device. When the C++ methods are encountered, device-specific versions of the routines are created that can be called from a target region.

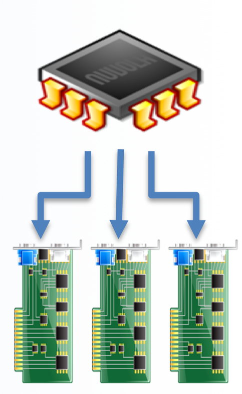
Deep copy is performed through the use of target enter data (lines 43 and 44) by first mapping the class and then the individual class members. Computation is performed on the device (line 46). After computation is over, the data is copy back to the host (line 52).



Simple Test Cases

```
1  int test_map_device() {
2    int num_dev = omp_get_num_devices(), sum[num_dev], errors = 0;
3    int* h_matrix = (int*) malloc(num_dev*N*sizeof(int));
4
5    for (int dev = 0; dev < num_dev; ++dev) {
6  #pragma omp target data map(from: h_matrix[dev*N:N]) device(dev)
7      {
8  #pragma omp target map(from: h_matrix[dev*N:N]) device(dev)
9        {
10         for (int i = 0; i < N; ++i)
11           h_matrix[dev*N + i] = dev;
12       } // end target
13     } // end target data
14   }
15
16   // checking results
17   errors = 0;
18   for (int dev = 0; dev < num_dev; ++dev) {
19     sum[dev] = h_matrix[dev*N + 0];
20     for (int i = 1; i < N; ++i)
21       sum[dev] += h_matrix[dev*N + i];
22     errors |= (dev * N != sum[dev]);
23   }
24
25   return errors;
26 }
```

Offloading Multiple devices:
Each row of the matrix to each of the available devices. Use the *device* clause to select a device for data movement and computation. Target data region maps a portion of the matrix to each device (line 6). Target region does the computation (line 8).

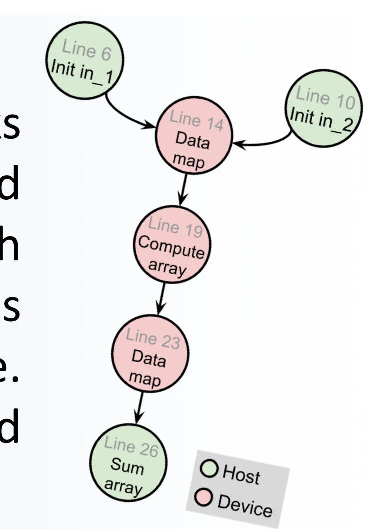


```
1  class B {
2  public:
3    static double VAR;
4    B() {}
5    static void modify(double* res) {
6  #pragma omp target map(tofrom: res[0:1])
7      {
8        *res = B::VAR;
9      }
10   }
11 };
12 double B::VAR = 1.0;
13
14 int test_static() {
15   int errors = 0;
16   double exp = 1.0, res = 0.0;
17
18   B::modify(&res);
19
20   errors = res != exp;
21   return errors;
22 }
```

Mapping static attribute of a class:
The unique value of the static VAR is default mapped inside the method of a class with a target region (lines 6-9). An OpenMP 4.5 capable compiler should capture the static variable (VAR) and map it to and from the device.

```
1  double sum = 0.0;
2  double* h_array = (double *) malloc(N * sizeof(double));
3  double* in_1 = (double *) malloc(N * sizeof(double));
4  double* in_2 = (double *) malloc(N * sizeof(double));
5
6  #pragma omp task depend(out: in_1) shared(in_1)
7  { for (int i = 0; i < N; ++i)
8      in_1[i] = 1; }
9
10 #pragma omp task depend(out: in_2) shared(in_2)
11 { for (int i = 0; i < N; ++i)
12     in_2[i] = 2; }
13
14 #pragma omp target enter data nowait \
15   map(alloc: h_array[0:N]) map(to: in_1[0:N]) \
16   map(to: in_2[0:N]) depend(out: h_array) \
17   depend(in: in_1) depend(in: in_2)
18
19 #pragma omp target nowait depend(inout: h_array)
20 { for (int i = 0; i < N; ++i)
21     h_array[i] = in_1[i]*in_2[i]; }
22
23 #pragma omp target exit data nowait \
24   map(from: h_array[0:N]) depend(inout: h_array)
25
26 #pragma omp task depend(in: h_array) \
27   shared(sum, h_array)
28 { for (int i = 0; i < N; ++i)
29     sum += h_array[i]; }
30
31 #pragma omp taskwait
32 errors = 2.0*N != sum;
```

Task dependencies:
Task graph composed of host tasks and target tasks that have in and out dependencies between each other. Asynchronous behavior is specified using the *nowait* clause. Data map tasks are separated from computation tasks.



Performance evaluation

Methodology

```
OMPVV_INIT_TEST;
for (i = 0; i < NUM_REP; i ++) {
  OMPVV_START_TIMER;
  #pragma omp ...
  OMPVV_TEST_LOAD; // if necessary
  OMPVV_STOP_TIMER;
  OMPVV_REGISTER_TEST;
}
OMPVV_PRINT_RESULT;
```

Measuring runtime overhead with multiple executions of different openMP directives and clauses. Each experiment consist of 1002 runs, removing the max and min and taking the average value.

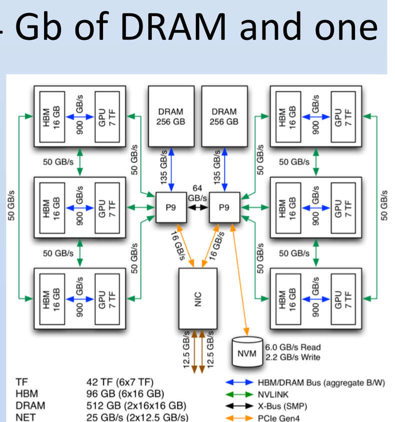
Testing systems

Two different systems and 5 different compilers and versions were tested. Summitdev features IBM S822LC nodes with POWER8 processors. Each node has a total of 160 hardware threads, 256 Gb of DRAM and as target devices, 4 NVIDIA Tesla P100 GPUs. The second system is an in-house cluster where each node features two Intel(R) Xeon(R) CPU E5-2670 with 32 Hardware threads, 64 Gb of DRAM and one NVIDIA K20.

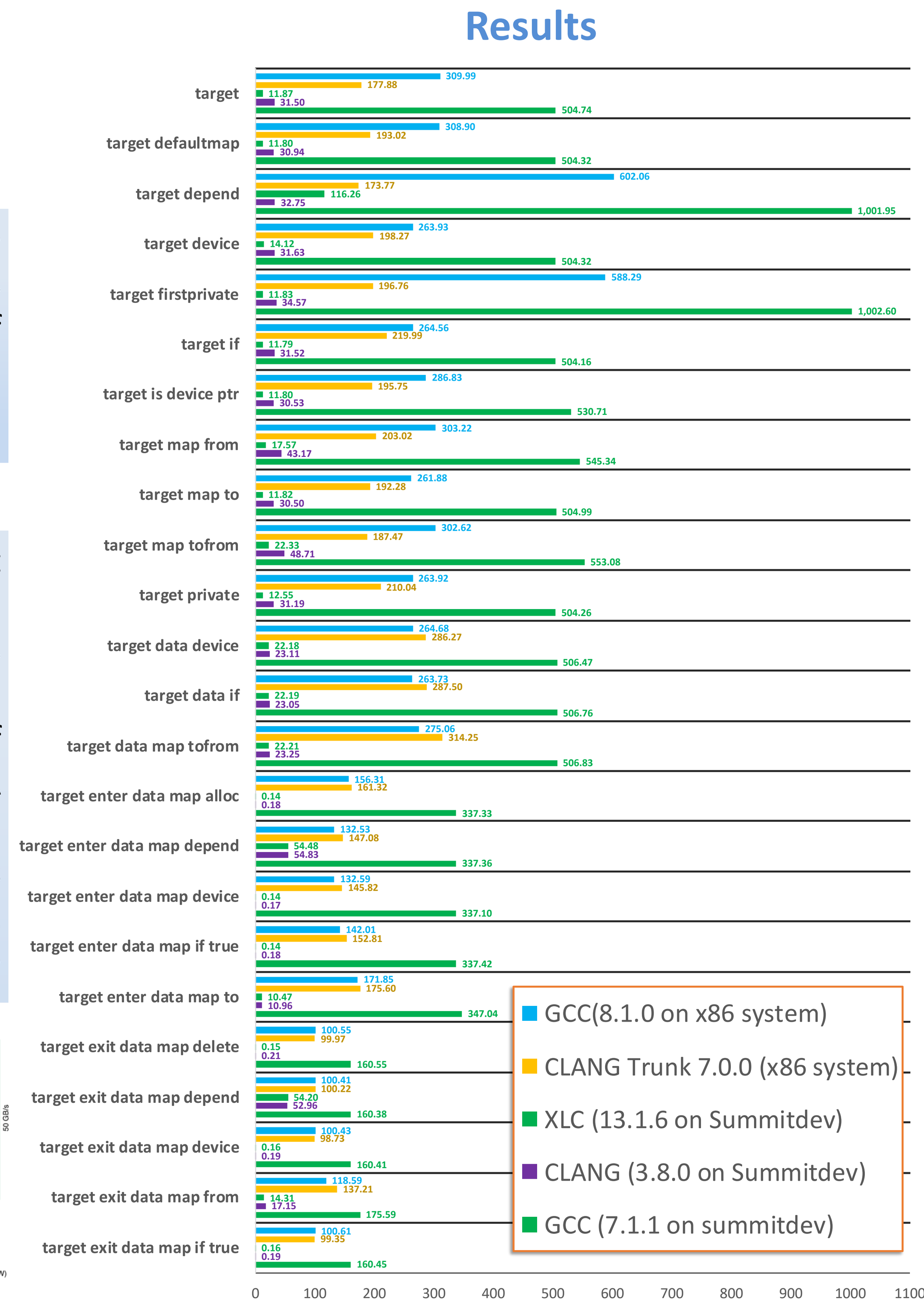
Compilers

Summitdev:
- GCC 7.1.1
- Clang 3.8.0
- XLC 13.1.6

In house cluster:
- GCC 8.1.0
- Clang 7.0.0 (Trunk version)



Results



Specification coverage evaluation

We are currently developing a test suite to asses the level of coverage of the OpenMP 4.5 specifications by the different compiler implementations. We have put together a methodology that guarantees full coverage of the specification as well as correct test implementation. We currently have released over 64 tests and we are currently in the process of releasing 33 more tests that are under review.

Systems

System	Model	Processors	Cores/node	Threads/node	Memory	Accelerator	Compilers
Titan	Cray XK7	AMD Opteron 6274	16	16	32 GB	1 NVIDIA K20X	CCE 8.7.2
Summitdev	IBM S822LC	2x Power8	20	160	256 GB	4 NVIDIA P100	GCC 7.1.1 Clang 3.8.0 XLC 13.1.6
Summit	IBM AC922	2x Power9	42	168	512 GB	6 NVIDIA V100	Clang 3.8.0 XLC 13.1.7

Results summary

	Summitdev					Summit		Titan
OpenMP 4.5 Feature	GCC 7.1.1	gfortran 7.1.1	Clang 3.8.0	XLC 13.1.6	XLF 15.1.7	Clang 3.8.0	XLC 13.1.7	CCE 8.7.2
target	14/14	13/13	14/14	13/14	12/13	12/14	11/14	13/14
target data	5/6	4/4	6/6	6/6	2/4	6/6	6/6	3/6
target enter/exit data	6/7	-	6/7	6/7	-	6/7	6/7	5/7
target enter data	6/7	-	6/7	6/7	-	6/7	6/7	5/7
target update	5/5	-	5/5	4/5	-	5/5	4/5	4/5
target teams distribute	10/11	-	8/11	10/11	-	-	-	9/11
target teams distribute parallel for	13/14	-	11/14	11/14	-	-	-	10/14

Visit our Website