Static Analysis of OpenMP data mapping for target offloading

Prithayan Barua,
Vivek Sarkar

Georgia Institute of Technology

Acknowledgements

Shirako Jun, Tsang Whitney, Paudel Jeeva, Chen Wang
OMPSan: Static Verification of OpenMP's Data Mapping Constructs.
IWOMP 2019

Outline

1. Introduction
   - OpenMP Target Offloading
2. Our Solution
   - Basic Idea
   - Analysis
   - Interpret OpenMP Clauses
3. Evaluation
   - Example Analysis
   - Conclusion
   - Experiment Results
4. Conclusion

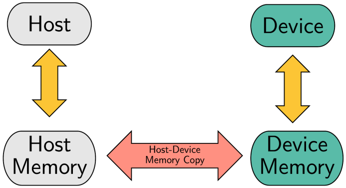
Outline

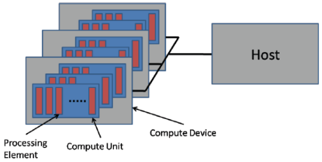
1. Introduction
   - OpenMP Target Offloading
2. Our Solution
   - Basic Idea
   - Analysis
   - Interpret OpenMP Clauses
3. Evaluation
   - Example Analysis
   - Conclusion
   - Experiment Results
4. Conclusion

Programming Heterogeneous Systems using OpenMP

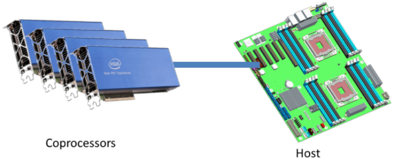
Programming Model

- Host can offload computations to target devices
- Each target device has a corresponding data environment
- Host can update the data between host and devices using *data mapping* clauses





Host and GPUs



Host and Co-processors

Using OpenMP for Target offloading

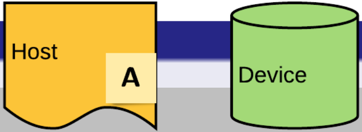
Example 1, How to ofload computations

```
#define N 10
int A[N], sum=0;
#pragma omp target data map(tofrom:A[0:N])
{
    #pragma omp target
    for(int i=0; i<N; i++) {
        A[i]=i;
    }
    #pragma omp target reduction(+:sum)
    for(int i=0; i<N; i++) {
        sum += A[i];
    }
}
```

generate a *target task*,
Map variables to a device data environment and
Execute the enclosed block of code on that device.

Semantics of *target data map*

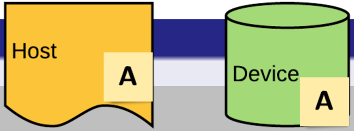
Example 1, L2




```
        #define N 10
L2:     ▷ int A[N], sum=0;
        #pragma omp target data map(tofrom:A[0:N])
L4:     {
            #pragma omp target
L7:         for(int i=0; i<N; i++) {
L8:             A[i]=i;
L9:         }
            #pragma omp target reduction(+:sum)
L11:        for(int i=0; i<N; i++) {
L12:            sum += A[i];
L13:        }
L14:    }
```

Semantics of *target 2*

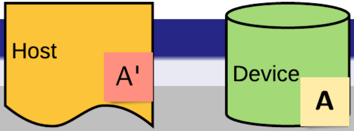
Example 1, L4




```
              #define N 10
L2:           int A[N], sum=0;
              #pragma omp target data map(tofrom:A[0:N])
L4:         ▷ {// Copy 'A[0:N]' to device.
                  #pragma omp target
L7:               for(int i=0; i<N; i++) {
L8:                   A[i]=i;
L9:               }
                  #pragma omp target reduction(+:sum)
L11:              for(int i=0; i<N; i++) {
L12:                  sum += A[i];
L13:              }
L14:          }//Copy 'A[0:N]' from device to host.
```

Semantics of *target*

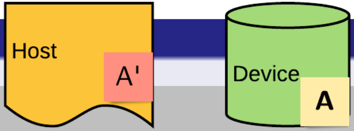
Example 1, L8




```
        #define N 10
L2:     int A[N], sum=0;
        #pragma omp target data map(tofrom:A[0:N])
L4:     {// Copy 'A[0:N]' to device.
            #pragma omp target
L7:         for(int i=0; i<N; i++) {// Execute on device
L8:      ▷      A[i]=i;
L9:         }// Leave 'A[0:N]' on device.
            #pragma omp target reduction(+:sum)
L11:        for(int i=0; i<N; i++) {// Execute on device.
L12:            sum += A[i];
L13:        }// Leave 'A[0:N]' and 'sum' on device.
L14:    }//Copy 'A[0:N]' from device to host.
```

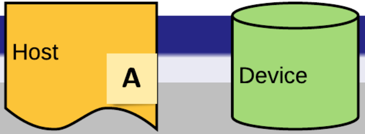
Semantics of *target*

Example 1, L12




```
     #define N 10
L2:  int A[N], sum=0;
     #pragma omp target data map(tofrom:A[0:N])
L4:  {// Copy 'A[0:N]' to device.
         #pragma omp target
L7:      for(int i=0; i<N; i++) {// Execute on device
L8:          A[i]=i;
L9:      }// Leave 'A[0:N]' on device.
         #pragma omp target reduction(+:sum)
L11:     for(int i=0; i<N; i++) {// Execute on device.
L12:     ▷   sum += A[i];
L13:     }// Leave 'A[0:N]' and 'sum' on device.
L14: }//Copy 'A[0:N]' from device to host.
```

Semantics of *target data map*



Example 1, L14

```
     #define N 10
L2:  int A[N], sum=0;
     #pragma omp target data map(tofrom:A[0:N])
L4:  {// Copy 'A[0:N]' to device.
         #pragma omp target
L7:      for(int i=0; i<N; i++) {// Execute on device
L8:          A[i]=i;
L9:      }// Leave 'A[0:N]' on device.
         #pragma omp target reduction(+:sum)
L11:     for(int i=0; i<N; i++) {// Execute on device.
L12:         sum += A[i];
L13:     }// Leave 'A[0:N]' and 'sum' on device.
L14: ▷ }//Copy 'A[0:N]' from device to host.
```

Execute *L11:* loop on host

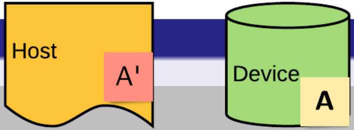
Example 2

```
     #define N 10
L2:  int A[N], sum=0;
     #pragma omp target data map(tofrom:A[0:N])
L4:  {
         #pragma omp target
L7:      for(int i=0; i<N; i++) {
L8:          A[i]=i;
L9:      }
         // #pragma omp target reduction(+:sum)
L11:     for(int i=0; i<N; i++) {
L12:         sum += A[i];
L13:     }
L14: }
```

Can we remove the pragma to execute the loop on host ?

Disaster !! Wrong Output

Example 2, L12




```
                #define N 10
L2:             int A[N], sum=0;
                #pragma omp target data map(tofrom:A[0:N])
L4:             {// Allocate 'A[0:N]' on device.
                    #pragma omp target
L7:                 for(int i=0; i<N; i++) {// Execute on device
L8:                     A[i]=i;
L9:                 }// Leave 'A[0:N]' on device.

L11:                for(int i=0; i<N; i++) {// Execute on host
L12:                  ▷ sum += A[i]; // Access host copy of stale 'A'!
L13:                }
L14:            }//Copy 'A[0:N]' from device to host.
```

But Why ?

Default Solution: OpenMP Specifications

OpenMP
Application Programming
Interface

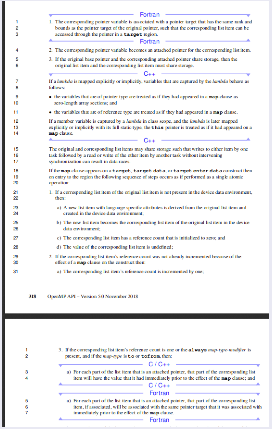
Version 5.0 November 2018

Copyright ©1997-2018 OpenMP Architecture Review Board.
Permission to copy without fee all or part of this material is granted, provided the OpenMP
Architecture Review Board copyright notice and the title of this document appear. Notice is

xiv OpenMP API – Version 5.0 November 2018

Understanding the Data Map Usage

Data Map Specification



Fortran

1. The corresponding pointer variable is associated with a pointer target that has the same rank and bounds as the pointer target of the original pointer, such that the corresponding list item can be accessed through the pointer in a **target** region.

Fortran

2. The corresponding pointer variable becomes an attached pointer for the corresponding list item.
3. If the original base pointer and the corresponding attached pointer share storage, then the original list item and the corresponding list item must share storage.

C++

If a *lambda* is mapped explicitly or implicitly, variables that are captured by the *lambda* behave as follows:

- the variables that are of pointer type are treated as if they had appeared in a **map** clause as zero-length array sections; and
- the variables that are of reference type are treated as if they had appeared in a **map** clause.

If a member variable is captured by a *lambda* in class scope, and the *lambda* is later mapped explicitly or implicitly with its full static type, the **this** pointer is treated as if it had appeared on a **map** clause.

C++

The original and corresponding list items may share storage such that writes to either item by one task followed by a read or write of the other item by another task without intervening synchronization can result in data races.

If the **map** clause appears on a **target**, **target data**, or **target enter data** construct then on entry to the region the following sequence of steps occurs as if performed as a single atomic operation:

1. If a corresponding list item of the original list item is not present in the device data environment, then:
   a) A new list item with language-specific attributes is derived from the original list item and created in the device data environment;
   b) The new list item becomes the corresponding list item of the original list item in the device data environment;
   c) The corresponding list item has a reference count that is initialized to zero; and
   d) The value of the corresponding list item is undefined;
2. If the corresponding list item's reference count was not already incremented because of the effect of a **map** clause on the construct then:
   a) The corresponding list item's reference count is incremented by one;

318 OpenMP API – Version 5.0 November 2018

3. If the corresponding list item's reference count is one or the **always** *map-type-modifier* is present, and if the *map-type* is **to** or **tofrom**, then:

C / C++

   a) For each part of the list item that is an attached pointer, that part of the corresponding list item will have the value that it had immediately prior to the effect of the **map** clause; and

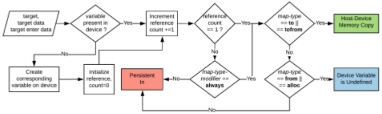
C / C++

Fortran

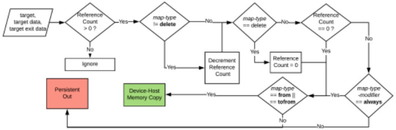
   a) For each part of the list item that is an attached pointer, that part of the corresponding list item, if associated, will be associated with the same pointer target that it was associated with immediately prior to the effect of the **map** clause.

Fortran

Our Flowchart to explain the Specification



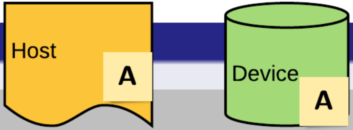
(a) Flowchart for Enter Device Environment



(b) Flowchart for Exit Device Environment

One possible fix

Example 3




```
#define N 10
int A[N], sum=0;
#pragma omp target data map(tofrom:A[0:N])
{
    #pragma omp target map(from:A[0:N])
    for(int i=0; i<N; i++) {
        A[i]=i;
    }
    ▷ #pragma omp target update from(A[0:N])
    for(int i=0; i<N; i++) {// Force Copy 'A[0:N]' to host.
      sum += A[i];
    }
}
```

Memory Optimization

Naive Jacobian

```
    while ( error > tol && iter < iter_max ) {
        error = 0.0;
#pragma omp target map(tofrom:Anew) map(tofrom:A) map(tofrom:error)
        for( int j = 1; j < n-1; j++)
            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 map(tofrom:Anew) map(tofrom:A)
        for( int j = 1; j < n-1; j++)
            for( int i = 1; i < m-1; i++ )
                A[j][i] = Anew[j][i];
        iter++;
    }
```

Memory Optimization

Remove Redundant Memory Copies

```
#pragma omp target data map(to:Anew) map(tofrom:A)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    #pragma omp target map(tofrom:error)
    for( int j = 1; j < n-1; j++)
        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
    for( int j = 1; j < n-1; j++)
        for( int i = 1; i < m-1; i++ )
            A[j][i] = Anew[j][i];
    iter++;
}
```

Motivation

- OpenMP is a widely used programming model for offloading computations from hosts to accelerators, industry standard across hardware vendors !
- Optimal or even correct usage of OpenMP data mapping constructs can be non-trivial and error-prone
- Enable the compiler to analyze the OpenMP program to help the developer with analysis reports, errors and warnings

Outline

Our Idea

Assumptions

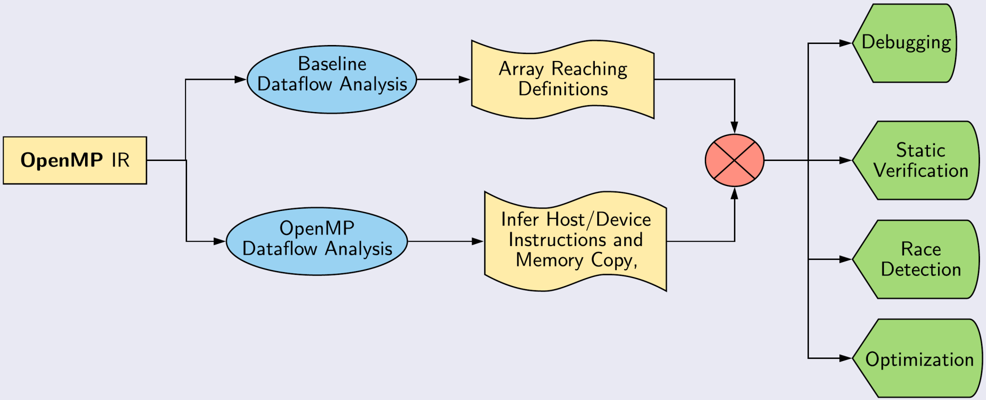
- Serial elision property: ***OpenMP program is expected to yield the same results when enabling or disabling OpenMP constructs***
- Dataflow information of the OpenMP and baseline sequential code must be the same

Claim

- We can use a static dataflow analysis, that compares the Array reaching definitions information of the OpenMP program with the Baseline to detect anomalies

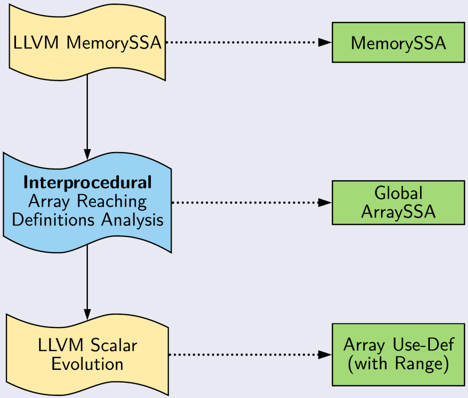
Our Solution

Overview of the System



Array Reaching Definition Analysis

Baseline Analysis based on Array SSA



Dataflow Equations for Array Reaching Definitions Analysis

- At, basic block B,

$$ReachingDef(B) = \bigcup_{P \in Pred(B)} ReachingDefOut(P)$$

$$ReachingDefOut(B) = ReachingDef(B) \cup GenDefs(B)$$

- At, Memory Access M,

$$ReachingDefAt(M) = Filter(M, \{ReachingDef(B) \cup GenDefs(M)\})$$

$$Filter(M, S) = \forall_{(X \in S | Alias(X,M) == true)} \{X\}$$

- For a Function, $Func$,

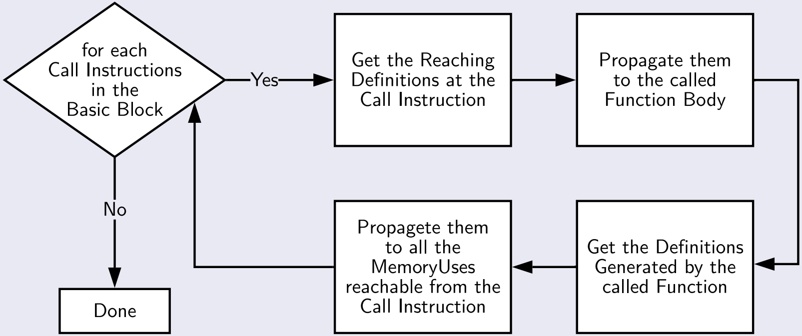
$$GeneratedDefFunction(Func) = \bigcup_{R \in return\ \ instructions} ReachingDefAt(R)$$

Interprocedural Analysis

Algorithm Summary

- Input: Reaching Definitions at every call instruction, and Definitions Generated by a function
- Output: Updated reaching definitions due to a call instruction

Propagate Reaching Defs across function call



OpenMP Run Time Library (RTL)

RTL Routines	Arguments
__tgt_target_data_begin (Initiate a device data environment)	int64_t device_id, int32_t num_args, void args_base, void args, int64_t args_size, int64_t args_maptype
__tgt_target_data_end (Close a device data environment)	– Same –
__tgt_target_data_update (Make a set of values consistent between host and device)	– Same –
__tgt_target (Begin/End data environment and launch target region execution)	–, void host_ptr, –
__tgt_target_teams (Specify Maximum teams and threads)	–, int32_t team_num, int32_t thread_limit

Clang Lowering to OpenMP Runtime Library

Example OpenMP target code

```
#pragma omp target map(tofrom:A[0:10])
for (i = 0 ; i < 10; i++) {
      A[i] = i;
  }
```

Corresponding LLVM pseudo-code with RTL

```
void **ArgsBase = {&A}
void **Args = {&A}
int64_t* ArgsSize = {40}
void **ArgsMapType = { OMP_TGT_MAPTYPE_TO | OMP_TGT_MAPTYPE_FROM }
call @__tgt_target(-1, HostAdr, 1, ArgsBase, Args, ArgsSize, ArgsMapType)
```

OpenMP IR

Baseline and Offloading calls

```
%23 = call i32 @__tgt_target_teams(i64 0,
        i8* @.__omp_offloading_801_15a4794_Mult_l29.region_id,
        i32 3, i8** %21, i8** %22,
        i64* getelementptr inbounds ([3 x i64],
        [3 x i64]* @.offload_sizes, i32 0, i32 0),
        i64* getelementptr inbounds ([3 x i64],
        [3 x i64]* @.offload_maptypes, i32 0, i32 0), i32 0, i32 0)
 %24 = icmp ne i32 %23, 0
 br i1 %24, label %omp_offload.failed, label %omp_offload.cont

omp_offload.failed:                                       ; preds = %entry
  call void @__omp_offloading_801_15a4794_Mult_l29(i32* %0, i32* %1, i32* %2) #6
  br label %omp_offload.cont
```

Outline

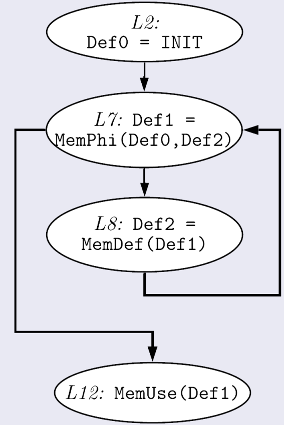
Georgia Tech
CREATING THE NEXT

Array Use-Def Chains (Step 1)

Baseline Sequential Code

```
#define N 10000
L2:int A[N], sum=0;
//#pragma omp target data map(from:A[0:N])
L4:{
//  #pragma omp target map(from:A[0:N])
L6:  {
L7:     for(int i=0; i<N; i++) {
L8:       A[i]=i;
L9:     }
L10:  }
L11:  for(int i=0; i<N; i++) {
L12:    sum += A[i];
L13:  }
L14:}
```

Array SSA for "A"



Interpret Semantics of *omp target* (Step 2)

With OpenMP Target

```
#define N 10000
L2:int A[N], sum=0;
//#pragma omp target data map(from:A[0:N])
L4:{
     #pragma omp target map(from:A[0:N])
L6:  {
L7:     for(int i=0; i<N; i++) { device
L8:       A[i]=i; device
L9:     } device
L10: }
L11:  for(int i=0; i<N; i++) { host
L12:     sum += A[i]; host
L13:  } host
L14:}
```

Classifying Execution Environment

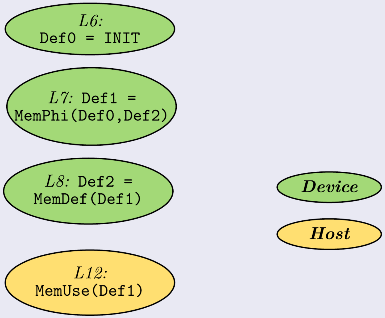
- Annotate each instruction, whether it executes on host or device, according to OpenMP specifications

Interpret Semantics of *omp target* (Step 2)

With OpenMP Target

```
#define N 10000
L2:int A[N], sum=0;
//#pragma omp target data map(from:A[0:N])
L4:{
     #pragma omp target map(from:A[0:N])
L6:  {
L7:    for(int i=0; i<N; i++) { device
L8:      A[i]=i; device
L9:    } device
L10: }
L11:  for(int i=0; i<N; i++) { host
L12:    sum += A[i]; host
L13:  } host
L14:}
```

ArraySSA Nodes with annotations

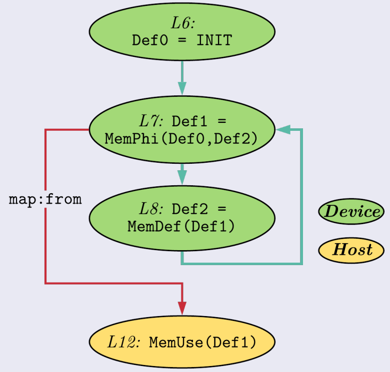


Infer Host/Device Memory Copies (Step 3)

With OpenMP Target

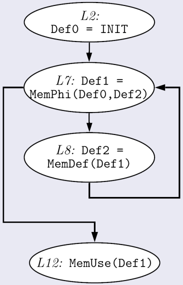
```
#define N 10000
L2:int A[N], sum=0;
//#pragma omp target data map(from:A[0:N])
L4:{
     #pragma omp target map(from:A[0:N])
L6:  {
L7:    for(int i=0; i<N; i++) { device
L8:      A[i]=i; device
L9:    } device
L10: }
L11:  for(int i=0; i<N; i++) { host
L12:    sum += A[i]; host
L13:  } host
L14:}
```

OpenMP Array SSA, with Memory Copies



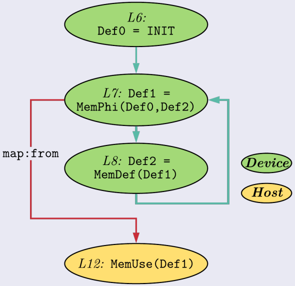
Anomaly Detection

Baseline Sequential Reaching Defs



- Reaching Defs(L12)={Def0,Def2}

OpenMP Reaching Defs



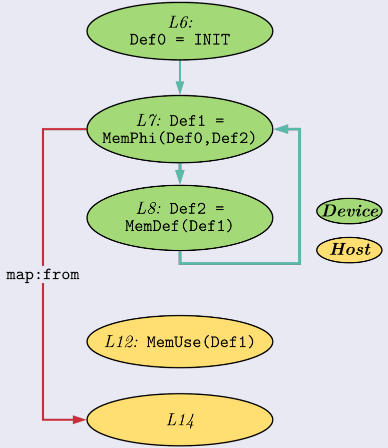
- Reaching Defs(L12)={Def0,Def2}

Incorrect Usage

Example 2,

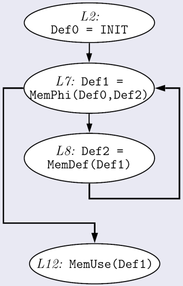
```
#define N 10000
L2:int A[N], sum=0;
#pragma omp target data map(from:A[0:N])
L4:{
  #pragma omp target map(from:A[0:N])
L6:  {
L7:    for(int i=0; i<N; i++) {
L8:      A[i]=i;
L9:    }
L10:  }
L11:  for(int i=0; i<N; i++) {
L12:    sum += A[i];
L13:  }
L14:}
```

Use-Def chains of OpenMP



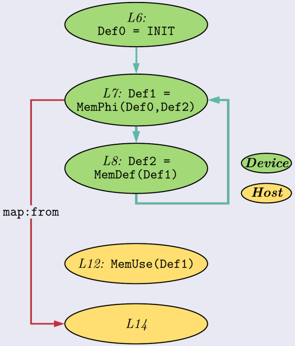
OpenMP Reaching Defs ≠ Baseline Reaching Defs

Baseline Sequential Reaching Defs



- Reaching Defs(L12)={Def0,Def2}

OpenMP Reaching Defs



- Reaching Defs(L12)={ }

Detecting Incorrect *target map* usage

1. For the baseline sequential program, Compute all the definitions reaching an Array use
2. Interpret the memory copies due to OpenMP target constructs according to OpenMP specifications and update the reaching definitions
3. Validate if all the original reaching definitions are still respected or not

Benchmark Results

DRACC File 22

```
int init(){
  for(int i=0; i<C; i++){
    for(int j=0; j<C; j++) {
L18:  b[j+i*C]=1;     }
    a[i]=1;
    c[i]=0;
  }}
int Mult(){
#pragma omp target map(to:a[0:C]) map(tofrom:c[0:C])\
                                map(alloc:b[0:C*C])
#pragma omp teams distribute parallel for
L32:  for(int i=0; i<C; i++){
       for(int j=0; j<C; j++)
L34:     c[i]+=b[j+i*C]*a[j];
    }
   }
```

```
int check(){
  bool test = false;
  for(int i=0; i<C; i++){
    if(c[i]!=C)
      test = true;
  }
}
int main(){
  a = malloc(C*sizeof(int));
  b = malloc(C*C*sizeof(int));
  c = malloc(C*sizeof(int));
  init();
  Mult();
  check();
  return 0;
  }
```

Benchmark Results

DRACC File 22

```
int init(){
  for(int i=0; i<C; i++){
    for(int j=0; j<C; j++) {
L18:  b[j+i*C]=1;     }
    a[i]=1;
    c[i]=0;
  }}
int Mult(){
#pragma omp target map(to:a[0:C]) map(tofrom:c[0:C])
                                map(alloc:b[0:C*C])
#pragma omp teams distribute parallel for
L32:  for(int i=0; i<C; i++){
       for(int j=0; j<C; j++)
L34:     c[i]+=b[j+i*C]*a[j];
      }
    }
```

OMPSan: Reported Error

ERROR Definition of :***b*** on Line:18 is not reachable to Line:34, Missing Clause:to:Line:32

Benchmark Results

DRACC File 23

```
int Mult(){
#pragma omp target map(to:a[0:C],b[0:C]) map(tofrom:c[0:C])
#pragma omp teams distribute parallel for
L32:  for(int i=0; i<C; i++){
       for(int j=0; j<C; j++)
L34:      c[i]+=b[j+i*C]*a[j];
    }
   }
int main(){
      a = malloc(C*sizeof(int));
L56:  b = malloc(C*C*sizeof(int));
      c = malloc(C*sizeof(int));
      init();
      Mult();
      check();
```

OMPSan: Reported Warning

WARNING Line:30 maps partial data:***b[0:50]***, but line 34 may access upto ***b[0:2500]***

Outline

Georgia Tech
CREATING THE NEXT

Limitations

- Supports statically and dynamically allocated array variables, but cannot handle dynamic data structures like linked lists
- Can only handle compile time constant array sections, and constant loop bounds.
- May report false positives for irregular array accesses, e.g., if a small section of the array is updated, our analysis may assume that the entire array was updated.
- May fail if Clang/LLVM introduces bugs while lowering OpenMP pragmas to the RTL calls in the LLVM IR.
- May report false positives, if the OpenMP program and baseline program do not have same output.

Summary

Static Analysis of OpenMP Programs

- Developed a static analysis tool to interpret the semantics of the OpenMP map clause, and deduce the data transfers introduced by the clause.
- Developed an interprocedural data flow analysis, to capture the reaching definitions information of Array variables.
- OmpSan: Validate if the data mapping in the OpenMP program respects the original reaching defs of the baseline sequential program.
- Ongoing: Optimization and Race detection

Georgia Tech
CREATING THE NEXT

Questions

Overview