OmpSan: Static Verification of OpenMP Data Mapping constructs

Prithayan Barua[1], Jun Shirako[1], Whitney Tsang[2], Jeeva Paudel[2], Wang Chen[2], and Vivek Sarkar[1]

[1] Georgia Institute of Technology
[2] IBM Toronto Laboratory

Abstract. **Keywords:** OpenMP Target Data Mapping · LLVM · Memory Management.

1 Introduction

The OpenMP is a widely used directive-based parallel programming model, which offers accelerator programming and supports heterogeneous computing systems with host CPUs and device accelerators (currently GPUs and FPGAs) from version 4.0 onwards. The accelerator-related features were largely extended in OpenMP 4.5: unstructured data mapping, asynchronous execution, and runtime routines for device memory management.

OMP 4.5 Target offloading and Data mapping The *omp target* directive is used to generate a target task that can be offloaded to a device, it also maps variables to the device data environment. The *omp target data* directive explicitly maps variables from a host environment to a device data environment.

On heterogeneous systems, the data movement between host and device is a common performance and energy efficiency bottleneck, as well as the major source of performance and correctness bugs. In the OpenMP accelerator model, host and device have their own memory space – i.e., data environment – and the data movement is supported by the explicit data copy via `map` clause. The optimal, or even correct specification of `map` clause is non-trivial error-prone because it requires element-wise dataflow analysis to identify which statement/instruction defines the value at given program point and variable/array element. Even given an existing application that uses the target offloading feature, understanding the data mapping behavior is nontrivial. Given a data map construct, its semantics depends on all the previous usages of the map construct, so it is context sensitive and the entire call sequence leading up to the construct can decide its behavior.

To address this issue as first-aid error detection, we propose a compile-time approach based on dataflow analysis. Assuming that enabling and disabling OpenMP result in the same output from the perspective of host/device data motion, our approach detects errors by the comparison of dataflow information

(reaching definitions via memory SSA) between the OpenMP and baseline code. We developed an LLVM-based implementation of our approach and evaluated the effectiveness using several case studies.

Our major contributions are summarized as follow.

- Implemented an algorithm to analyze the OpenMP runtime library calls inserted by clang in the LLVM IR, to infer the host/device memory copies according to OpenMP 4.5 semantics
- Develop a static analysis technique to validate if the host/device memory copies respect the original memory use-def relations.
- Output error/warnings on usage of data mapping constructs
- Debug information to understand how the map clause affects the host and device data environment.

The paper is organized as follows. section 2 provides certain motivating examples, that show common bugs and difficulties in usage of the *data map* construct. section 3 presents an overview of our approach to validate the usage of data mapping constructs. section 4 presents the LLVM implementation details, and section 5 presents the evaluation and some case studies.

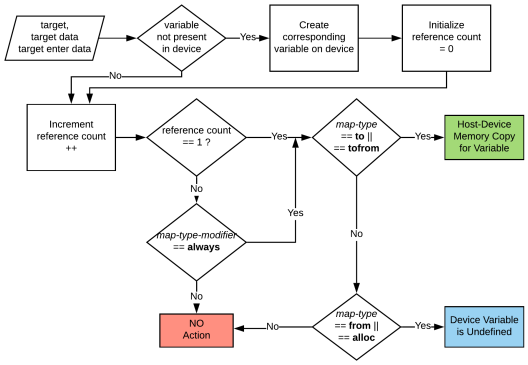
2 Motivating Examples

2.1 OpenMP 4.5 Map Semantics

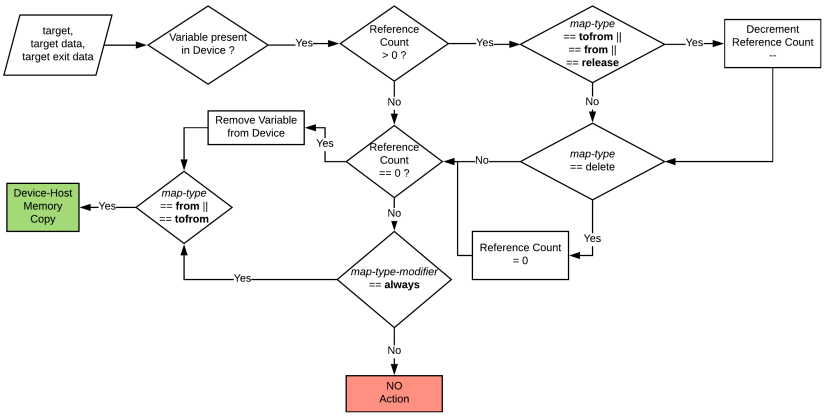
The OpenMP 4.5 spec specifies the semantics of the *map* clause, and Figure 1 illustrates how the spec interprets the data map constructs. The flowchart also motivates our argument that understanding the *map* clause is not trivial, it shows the complex set of rules defined in the OpenMP 4.5 spec, that is used to determine how to map a host variable to the corresponding list item in the device data environment. We make a simplifying assumption that the CPU host is the current task's data environment and the device is a GPU, and hence the mapping from the host to the device refers to the Device-Host and Host-device memory copy inserted because of the map clause. The spec uses reference count for a variable, to decide when to introduce a device/host memory copy. The host to device memory copy is introduced only when the reference count is incremented from 0 to 1 and the "to" attribute is present. Then the reference count keeps incrementing every time the a new device map environment is created. The reference count is decremented on encountering a "from" or "release" attribute, while exiting the data environment. Finally when the reference count is decremented to zero from 1, and the "from" attribute is present, the variable is copied back to the host from the device. Figure 2 shows an example state machine, to decide when to insert the memory copies.

2.2 Default Scalar Mapping

Our first example for a bug in user programs in Listing 2.1. In this example, the definition of "sum" on line 5 does not reach line 6, since the "sum" does not



(a) Flowchart for inserting Host-Device Memory Copy



(b) Flowchart for inserting Device-Host Memory Copy

Fig. 1: Flowchart to show how to interpret the map clause

have an explicit mapping and the default map for scalars is “fristprivate”. As Listing 2.2 shows, we need to use explicit map clause, to copy in and copy out the scalar “sum” from device.

Listing 2.1: Default scalar map

```
1 int A[N], sum=0, i;
2 #pragma omp target
3 #pragma omp teams distribute parallel for
        reduction(+:sum)
4   for(i=0; i<N; i++)
5     sum += A[i];
6   printf("\n%d",sum);
```

Listing 2.2: Explicit map

```
1 int A[N], sum=0;
2 #pragma omp target map(tofrom:sum)
3 #pragma omp teams distribute parallel for
        reduction(+:sum)
4   for( int i=0; i<N; i++)
5     sum += A[i];
6   printf("\n%d",sum);
```

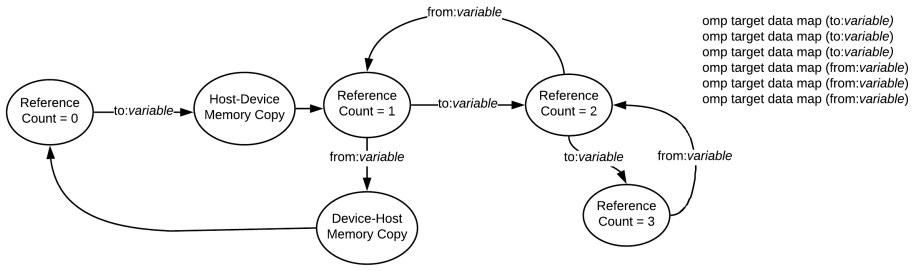


Fig. 2: State Machine for inserting Host/Device Memory Copies

2.3 Reference Count Issues

Listing 2.3: Usage of alloc vs from

```
1  int A[10], B[10];
2  for (int i =0 ; i < 10 ; i++)
3      A[i] = i;
4
5  #pragma omp target enter data map(to:A
       [0:10]) map(alloc:B[0:10])
6  #pragma omp target map(alloc:B[0:10])
7  for (int i = 0 ; i < 10; i++)
8      B[i] = A[i];
9  #pragma omp target exit data map(from:B
       [0:10])
10
11 for (int i = 0 ; i < 10; i++)
12     printf("%d",B[i]);
```

Listing 2.4: Correct Usage

```
1  int A[10], B[10];
2  for (int i =0 ; i < 10 ; i++)
3      A[i] = i;
4
5  #pragma omp target enter data map(to:A
       [0:10]) map(alloc:B[0:10])
6  #pragma omp target map(from:B[0:10])
7  for (int i = 0 ; i < 10; i++)
8      B[i] = A[i];
9  #pragma omp target exit data map(from:B
       [0:10])
10
11 for (int i = 0 ; i < 10; i++)
12     printf("%d",B[i]);
```

Listing 2.3 shows another example of fine details of the spec, that can be overlooked by a programmer. According to Figure 1 The reference count for variable “B” is set to 1 at line 5, as we enter a new data environment. Then we enter the data environment at line 6, which increments the reference count to 2. But as we exit the environment at line 8, because of the “alloc” attribute, the reference count of variable “B” is not decremented, so at line 8, reference count is 8, and line 9 ends the data environment, and has a “from“ attribute which decrements the reference count to 1. Since the reference count is not zero, the ”B“ is not mapped back to host, and line 12 accesses the stale data.

This bug points out the difficulty in interpreting a map construct independently. Imagine the same code and we are dealing with global variables, with the map clauses in different functions, in different source files. At that point it is almost impossible to debug this issue manually. Our static analysis tool can not only error out on such issues, but also the show debug information to help understand how each map construct is interpreted based on its context.

Listing 2.4 shows the fix for this bug, by using the ”from“ attribute on line 6.

Listing 2.5 shows another incorrect usage of map clause. The user declared the target data environment on line 3, with “A” mapped as “from”. According to the OpenMP 4.5 semantics, the map clause on line 3, will instantiate an

uninitialized version of array "A" on device, and also associate a reference count with it. The reference count will be set to 1, after the line 3. Now the map clause on the "target" construct, will have no affect on the device copy of "A", but still it will increment the reference count to 2 at line 6. At the exit of the offloaded loop, after line 7, the reference count is 2, hence the "from" clause on line 5 will not have any affect, other than decrementing the reference count to 1. Now the line 9, that is executed on the host, will not be reading the updated version of "A" from the device, line 7, because "A" was not mapped back to the host after line 7. On exit of the data map region at line 10, the reference count is decremented to 0, and only then the device copy of "A" is mapped back and copied to the host "A". This is because of the "from" map attribute on line 3. In this example, "from" attribute on line 5, has no affect. To fix this issue, we need to use the update clause as shown in Listing 2.6.

Listing 2.5: Reference Count

```
1  define N 100
2  int A[N], sum=0;
3  #pragma omp target data map(from:A[0:N])
4    {
5      #pragma omp target map(from:A[0:N])
6      for(int i=0; i<N; i++)
7        A[i]=i;
8      for(int i=0; i<N; i++)
9        sum += A[i];
10   }
```

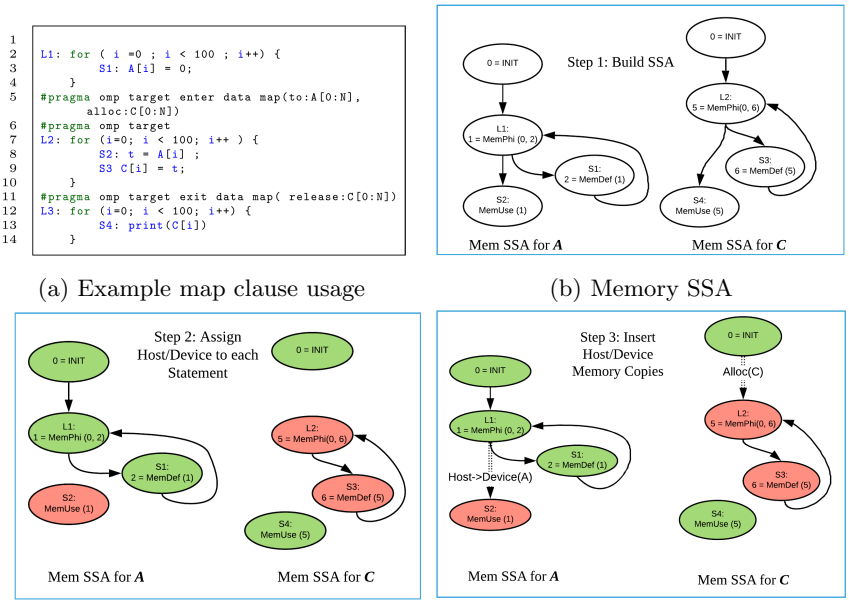
Listing 2.6: Update Clause

```
1
2  define N 100
3  int A[N], sum=0;
4  #pragma omp target data map(from:A[0:N])
5    {
6      #pragma omp target map(from:A[0:N])
7      for(int i=0; i<N; i++)
8        A[i]=i;
9      #pragma omp target update from(A[0:N])
10     for(int i=0; i<N; i++)
11       sum += A[i];
12   }
```

3 Our Approach

3.1 Memory SSA

Let us consider an example to illustrate our approach to verify the correctness of the *map* clauses used by a programmer. Figure 3a shows a small program fragment, to illustrate the usage of map clause. Figure 3b shows the memory SSA constructed from the program. We can see the use-def chains for the two memory variables A and C individually. The memory SSA clearly shows the uses and the reaching definitions for the corresponding memory variable. The next step is to interpret the target offload constructs used in the program. The target construct is used to offload a region of code onto a device. Figure 3c shows the interpretation of the target construct at line 6 of the program. The figure uses a color code, to illustrate that the red statements are in the device environment, while the green nodes are the host data environment. In this step, we also remove the memory use-def edges that are across two different environments. So for example, there is no edge between L1 and S2, since the L1 loop is executed on the host, while the S2 is on the device. The last step of our analysis is to interpret the *map* clause used in the program, and add special edges in the memory SSA graph, that corresponds to the host/device memory copy. This step follows the algorithm depicted in Figure 1. So because of the to map clause on line 5, there is an edge between L1 and S2, which respects the reaching definitions for S2 as per Figure 3b. Also, the Alloc edge from init node to L2, signifies the alloc



(a) Example map clause usage (b) Memory SSA

(c) Classify Host/Device Regions (d) Host/Device Memory Copies

Fig. 3: Steps to Validate map clause

map clause for memory variable C. Now, for every Memory Use node, we can validate if there is an edge from its corresponding memory def. Otherwise, the *map* clauses were used incorrectly in the user program. So for example, there is still a missing edge between Memory use at statement S4. This means map clauses are wrong and S4 is not reading the appropriate value.

3.2 Array Sections

Next, we look at how to validate the array sections used in the map clause. We modify the example, Figure 3a, with the following map clauses,

```
#pragma omp target enter data map(to:A[0:50], alloc:C[0:100])
#pragma omp target exit data map(from:C[0:100])
```

We start with the memory SSA graphs for each memory variable. Each memory def and Memory use corresponds to a statement in the user program. We can analyze the corresponding statement to figure out the index expressions, and the range of values that the expression evaluates to, as shown in Figure 4. For example, the index expression at statement S1, has a minimum value of 0 and a maximum value of 100. If the range of the index expression cannot be evaluated statically, then we approximate the range to the size of the memory variable. The size of an array variable is known statically, and the size of dynamically

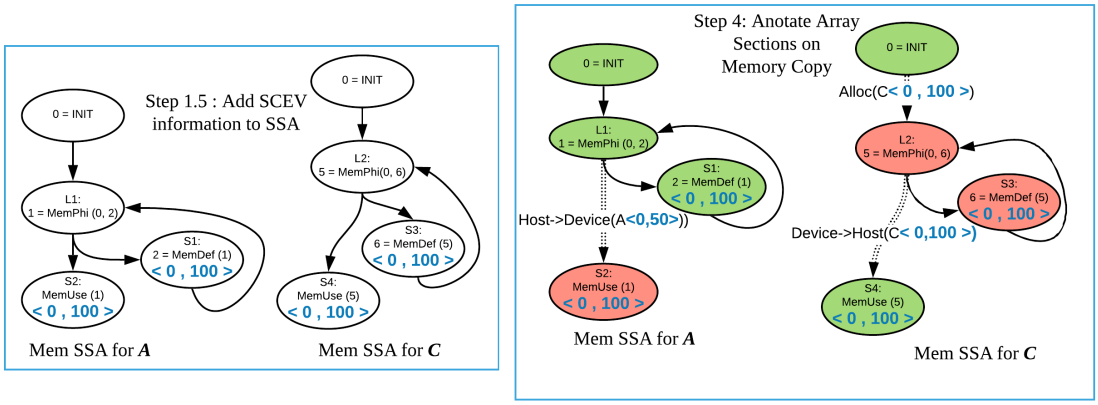


Fig. 4: Array Sections Analysis

allocated memory variables can be obtained by parsing the corresponding malloc instruction. With these annotations, in place, we have a conservative estimate of the range of locations that each memory access refers to. Next step is to interpret the array sections used in the map clauses by the user. It is possible only if the user used compile time constants to specify the array sections in the map clause, as shown in Figure 4. Each edge between two nodes of different data environments is now annotated with a range. For each such edge, we validate that the corresponding use range is a subset of the range of the edge. Since this analysis is an overapproximation, the violation of this property need not always be an error. Hence we classify this property as a warning.

4 Implementation

We have implemented our framework in LLVM 7.0, which has the OpenMP 4.5 implementation. The OpenMP constructs are lowered to runtime calls in Clang, so in the LLVM IR we only see calls to the OpenMP runtime. There are several disadvantages of this approach especially with respect to our analysis. Firstly the region of code that needs to be offloaded to a device, is opaque since it is moved to separate functions. These functions are in turn called from the OpenMP runtime library. As a result, we cannot perform a global data flow analysis for the memory use-def information of the offloaded region. To get around this limitation, we implemented a two-pass approach. Firstly we compile the OpenMP program with the flag that enables parsing the OpenMP constructs, and then again without the flag, such that Clang ignores the OpenMP constructs and generates the baseline LLVM IR. During the OpenMP compilation pass, Clang calls our analysis pass, that parses the runtime library calls and generates a csv file that records all the user specified “target map” clauses, as explained in subsection 4.1. During the second pass, we perform the whole program context and flow sensitive data

flow analysis, to construct the Memory use-def chains, subsection 4.2. Then this pass validates if the "target map" information generated by the previous pass, respects all the Memory use def relations.

4.1 Interpreting OpenMP pragmas

To enable parsing the OpenMp pragmas, clang needs the following flag.

```
-fopenmp=libomp -omptargets=<>
```

The offload mechanism used by clang is to generate calls to a runtime library(RTL) whenever "target" directives are encountered. The offload library implements the routines shown in Table 1. In the LLVM IR, whenever we encounter a call to one of these RTL routines, we parse the arguments of the functions, and extract the relevant information from them. Table 2 lists the arguments that are relevant to interpret the semantics of the *map* clause.

Table 1: Target Runtime Library Routines

RTL Routines	Arguments

__tgt_target_data_begin :: Initiate a device data environment	`int32_t device_id, int32_t num_args`
	`void** args_base, void** args,`
	`int64_t *args_size, int32_t *args_maptype`

__tgt_target_data_end :: Close a device data environment	`int32_t device_id, int32_t num_args`
	`void** args_base, void** args,`
	`int64_t *args_size, int32_t *args_maptype`

__tgt_target_data_update :: Make a set of values consistent between host and device	`int32_t device_id, int32_t num_args`
	`void** args_base, void** args,`
	`int64_t *args_size, int32_t *args_maptype`

__tgt_target :: Begin data environment, launch target region execution and end device environment	`int32_t device_id, void *host_addr,`
	`int32_t num_args`
	`void** args_base, void** args,`
	`int64_t *args_size, int32_t *args_maptype`

__tgt_target_teams :: Same as above, also specify number of teams and threads	`int32_t device_id, void *host_addr,`
	`int32_t num_args, void** args_base,`
	`void** args, int64_t *args_size,`
	`int32_t *args_maptype,`
	`int64_t num_teams, int32_t thread_limit`

Table 2: Target Runtime Library Routine Arguments Explanation

Argument	Explanation
device_id	Uniquely Identify the target
num_args	Number of data pointers that require a mapping
*void** *args*	Pointer to an array with *num_args* arguments, whose elements point to the first byte of the array section that needs to be mapped
*int64_t** *args_size*	Pointer to an array with *num_args* arguments, whose elements contain the size in bytes of the array section to be mapped
*void** *args_base*	Pointer to an array with *num_args* arguments, whose elements point to base address and differs from *args* if an array section does not start at 0
*void * * args_maptype*	Pointer to an array with *num_args* arguments, whose elements contain the required map atribute specified by the enum Table 3

Table 3: Target Runtime Library Map Type Attribute Enum

Enum Type	Map Clause
OMP_TGT_MAPTYPE_ALLOC	alloc
OMP_TGT_MAPTYPE_TO	to
OMP_TGT_MAPTYPE_FROM	from
OMP_TGT_MAPTYPE_ALWAYS	always
OMP_TGT_MAPTYPE_RELEASE	release
OMP_TGT_MAPTYPE_DELETE	delete
OMP_TGT_MAPTYPE_POINTER	map a pointer instead of array

Listing 4.1: Example map clause

```
1 #pragma omp target
2     map(tofrom:A[0:10])
3     for (i = 0 ; i < 10; i++)
4       A[i] = i;
```

Listing 4.2: Pseudocode for LLVM IR with RTL calls

```
1 void **ArgsBase = {&A}
2 void **Args = {&A}
3 int64_t* ArgsSize = {400}
4 void **ArgsMapType = { OMP_TGT_MAPTYPE_TO
        | OMP_TGT_MAPTYPE_FROM }
5 call @__tgt_target
6 (-1, HostAdr, 1, ArgsBase,
7 Args, ArgsSize, ArgsMapType)
```

Listing 4.1 shows a very simple user program, with the target data map clause. Listing 4.2 shows the corresponding LLVM IR, after clang introduces the runtime calls. Line 5 is one of the RTLs from the Table 1, and we parse the arguments of this call according to the Table 2. The 3rd argument to the call at line Listing 4.2 is 1, that means there is only item in the map clause. Line 1, that is the value loaded into *ArgsBase* is used to get the memory variable that is being mapped. Line 3, *ArgsSize* gives the end of the corresponding array section, starting from *ArgsBase*. Line 4, *ArgsMapType*, gives the map attribute used by the programmer, that is "tofrom".

We wrote an LLVM pass, that analyzes every such RTL call, and tracks the value of each of its arguments, as explained above. Once we have this information, we use the algorithm of Figure 1 to interpret the data mapping semantics of each clause. The data mapping semantics can be classified into following categories,

- CopyIn: A memory copy is introduced from the host to the device environment.
- CopyOut: A memory copy is introduced from the device to the host environment.
- PersistentOut: A device memory variable is not deleted, it is persistent on the device, and available to the subsequent device data environment.
- PersistentIn: The memory variable is available on entry to the device data environment, from the last device invocation.

To illustrate the above classification, consider the example, Listing 4.3. Line 6 of the example, creates a data environment, with "to" mapping for $A[0:10]$, while $B[0:10]$ is allocated on the device. This is illustrated in the first row of Table 4, the RTL signifies start of device data environment, line begin and end refer to the same line, with $A[0:10]$ as the copy in, and $B[0:10]$ as Alloc. Next target map clause on line 7, specifies the offloaded region of code, along with a map clause. According to the OpenMP 4.5 semantics, as shown in the table, both $A[0:10]$ and $B[0:10]$, are persistent in, because of the "enter data map" clause on line 6. While $A[0:10]$ is copy out, $B[0:10]$ is still persistent out, and copied out only because of the "exit data map" clause on line 10.

Listing 4.3: Example OpenMP map construct

```
1  int main(){
2    int A[10], B[10];
3    for (int i =0 ; i < 10 ; i++)
4      A[i] = i;
5
6  #pragma omp target enter data map(to:A[0:10]) map(alloc:B[0:10])
7  #pragma omp target map(from:B[0:10])
8    for (int i = 0 ; i < 10; i++)
9      B[i] = A[i];
10 #pragma omp target exit data map(from:B[0:10])
11
12   for (int i = 0 ; i < 10; i++)
13     printf("%d",B[i]);
14
15     return 0;
16 }
```

Table 4: Output Data mapping for Listing 4.3

RTL name	Region Line Begin	Region Line end	Copy In	Persistent In	Copy Out	Persistent out	Alloc
__tgt_target_data_begin	6	6	A[0:10]				B[0:10]
__tgt_target	7	10		A[0:10], B[0:10]	A[0:10]	B[0:10]	
__tgt_target_data_end	10	10			B[0:10]		

4.2 Baseline Memory Use Def Analysis

Once we have the information regarding the memory copies introduced by the map clause we construct the Memory SSA of the program. In the LLVM IR, to enable such an analysis, we have to compile the program by ignoring the target map clauses. We also perform inlining of all user functions to perform a context-sensitive analysis. So, as far as our analysis is concerned, only load and store instructions can modify memory, and all call instructions are inlined.

Memory SSA LLVM has an analysis called the MemorySSA[2]. It is a relatively cheap analysis that provides an SSA based form for memory use-def and def-use chains. LLVM MemorySSA is a virtual IR, which maps "Instructions" to "MemoryAccesses", which is one of three kinds, "MemoryPhi", "MemoryUse", "MemoryDef". Viewing them in terms of heap versions is helpful. Operands of any "MemoryAccess" are a version of the heap before that operation, and if the access can modify the heap, then it produces a value, which is the new version of the heap after the operation. Another important fact is that the "MemoryPhi" merges may-reach definitions. Figure 5 shows an example, with Figure 5b as the simplified MemorySSA. We have simplified this example, to make it relevant to our context. *LiveonEntry* is a special "MemoryDef" that dominates every "MemoryAccess" within a function, and implies that the memory is either undefined or defined before the function begins. The "MemoryAccess" $2 = MemoryPhi(1, 3)$ corresponds to the for loop on line 3. There are two versions of the heap reaching it, 1 from the entry block, and 3 from the back edge, and it produces a new heap version 2. Next, $3 = MemoryDef(2)$, notes that there is a store instruction which clobbers the heap version 2, and generates heap 3. This instruction corresponds to line 8, in the user program. The next "MemoryAccess", $4 = MemoryPhi(2, 5)$, corresponds to the for loop at line 6. Again the clobbering accesses that reach it are 2 from the previous for loop and 5, from its own body. The load of memory A on line 7, corresponds to the $MemoryUse(4)$, that notes that the last instruction that could clobber this read is MemoryAccess 4. Then, $5 = MemoryDef(4)$ clobbers the heap, to generate heap version 5. This corresponds to the write to array B on line 7. This is an important example of how LLVM deliberately trades off precision for speed. It considers the memory variables as disjoint partitions of the heap, but instead of trying to disambiguate aliasing, in this example, both stores/MemoryDefs clobber the same heap partition. Finally, the read of B on line 10, corresponds to $MemoryUse(4)$, with the heap version 4, reaching this load. The MemorySSA of Figure 5b is too conservative since every store instruction generates a new version of the heap. However, we can walk over this MemorySSA, and disambiguate the memory variable that the store instruction refers to. So, for any store instruction, for example, line 19, we can analyze the LLVM IR, and trace the value that the store instruction refers to. In this example, line 18, refers to memory variable B, and hence clearly, the $5 = MemoryDef(4)$ does not clobber memory variable A. We perform an analysis on the LLVM IR, which tracks the set of memory variables that each LLVM load/store instruction refers to. We developed a

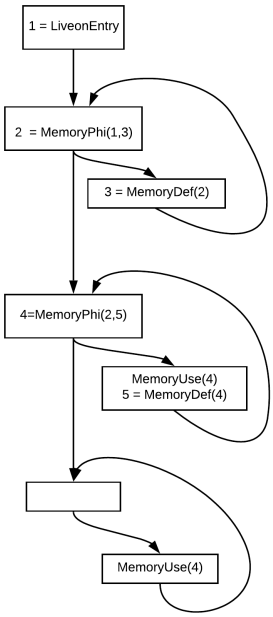
```c
int main(){
  int A[10], B[10];
  for (int i =0 ; i < 10 ; i++) {
    /*
      %arrayidx = getelementptr %A, 0, %idxprom
      store %i.0, %arrayidx,
    */
    A[i] = i;
  }

  for (int i = 0 ; i < 10; i++) {
    /*
      %arrayidx7 = getelementptr %A, 0, %idxprom6
      %2 = load %arrayidx7
    */
    int t =  A[i];
    /*
      %arrayidx9 = getelementptr %B, 0, %idxprom8
      store %2, %arrayidx9
    */
    B[i] = t
  }

  for (int i = 0 ; i < 10; i++) {
    /*
      %arrayidx19 = getelementptr %B, 0, %idxprom18
      %3 = load %arrayidx19
    */
    printf("%d",B[i]);
  }

    return 0;
}
```

(a) Code Fragment



(b) LLVM Memory SSA Virtual IR

Fig. 5: LLVM MemorySSA Example

context-sensitive and flow-sensitive iterative data flow analysis that associates each MemoryDef/MemoryUse with a set of memory variables. If the set is a singleton set, then it is a *Must* access, otherwise, it is a *May* access.

4.3 Validation of data mapping

After the data flow analysis on the memory SSA, we have the memory use-def chains. Also, the analysis of subsection 4.1 interprets the memory copies introduced by the omp map clauses. Now as explained in section section 3, for every pair of Memory use-defs, we verify if the map clause respects that relation. We use the LLVM "OptimizationRemarkEmitter" to output the result of our analysis. As we had classified each memory access as must/may, we classify the must violations as Errors, and the may violations as warnings. We can prove that the Must violations, found out by our analysis are real bugs in memory mapping clause in the user program.

4.4 SCEV Analysis, Array Sections

LLVM Scalar Evolution is a very powerful technique that can be used to analyze the change in the value of scalar variables over iterations of a loop. We can use the SCEV analysis to represent the loop induction variables as chain of recurrences. This mathematical representation can then be used to analyze the variables used to index into memory operations. The SCEV representation in Figure 5a for variable i on line 16 is " $\{0, +, 1\} < \%for.Line11 >$ ".

- The first term of the SCEV is the initial value
- Second term is a mathematical step operation, in this case it is the increment operator
- Third term is the value which is used to apply the step operation, in this case it is increment by 1, it can also be another SCEV to express chain of recurrences
- SCEV also contains the corresponding enclosing for loops

For our analysis, we need the range of memory index expression, that is the minimum and maximum values, of the index expression of any memory address. The SCEV analysis can be queried to get the maximum number of iterations of a loop. This value can be used to evaluate the maximum value of the SCEV expression. We have implemented an algorithm in our analysis pass, that given a load/store, computes the minimum and maximum values of the corresponding index into the memory access. This minimum and maximum can be an expression in terms of program variables. We use this analysis to extract the array sections accessed by any memory load/store.

There are three main cases for the result of SCEV analysis

1. The upper and lower bounds are compile time constants
2. The upper and lower bounds are expressions of program variables
3. SCEV is an unknown or cannot be evaluated

For our analysis, we approximate the case 2 and 3, with the maximum size of an array. If the memory load/store is from a static array, then the bounds of the array are compile time constants. Otherwise we parse the arguments of the call to memory alloc (malloc), to get the bounds of the corresponding memory variable.

So, the Array Sections analysis works only if the range of a memory access can be evaluated to compile time integer constants. We then use the analysis explained in subsection 3.2 to warn the user if incorrect array sections are used in the memory map clause. The warning can be of the following types

- Array section accessed on the device environment is larger than the array section mapped onto the device, that is out of bounds access on the device
- Array section mapped out of the device environment to the host, is smaller/subset of what the host environment accesses.

5 Evaluation and Case Studies

Listing 5.1: DRACC File 22

```
15 int init(){
16     for(int i=0; i<C; i++){
17         for(int j=0; j<C; j++){
18             b[j+i*C]=1;
19         }
20         a[i]=1;
21         c[i]=0;
22     }
23         return 0;
24 }
25
26
27 int Mult(){
28
29     #pragma omp target map(to:a[0:C]) map(
       tofrom:c[0:C]) map(alloc:b[0:C*C]) device
       (0)
30     {
31         #pragma omp teams
                   distribute parallel for
32         for(int i=0; i<C; i++){
33             for(int j=0; j<C; j++){
34                 c[i]+=b[j+i*C]*a[j];
```

Listing 5.2: DRACC File 23

```
28 int Mult(){
29
30     #pragma omp target map(to:a[0:C],b[0:C])
       map(tofrom:c[0:C]) device(0)
31     {
32         #pragma omp teams
                   distribute parallel for
33         for(int i=0; i<C; i++){
34             for(int j=0; j<C; j++){
35                 c[i]+=b[j+i*C]*a[j];
36             }
37         }
38     }
```

Listing 5.3: DRACC File 26

```
29     #pragma omp target
               enter data map(to:a[0:C],b[0:C*C
                   ],c[0:C]) device(0)
30     #pragma omp target device(0)
31     {
32         #pragma omp teams
                       distribute parallel for
33         for(int i=0; i<C; i++){
34             for(int j=0; j<C; j++){
35                 c[i]+=b[j+i*C]*a[j];
36             }
37         }
38     }
39     #pragma omp target exit
               data map(release:c[0:C])
               map(release:a[0:C],b[0:C*C])
                   device(0)
40     return 0;
41 }
42
43 int check(){
44     bool test = false;
45     for(int i=0; i<C; i++){
46         if(c[i]!=C){
```

Listing 5.4: DRACC File 30

```
19 int init(){
20     for(int i=0; i<C; i++){
21         for(int j=0; j<C; j++){
22             b[j+i*C]=1;
23         }
24         a[i]=1;
25         c[i]=0;
26     }
~
31 int Mult(){
32
33     #pragma omp target
           map(to:a[0:C],b[0:C*C]) map(from:c[0:
               C*C]) device(0)
34     {
35         #pragma omp teams
                   distribute parallel for
36         for(int i=0; i<C; i++){
37             for(int j=0; j<C; j++){
38                 c[i]+=b[j+i*C]*a[j];
```

We use the the DRACC benchmark [1] from Aachen University to evaluate our tool. Table 5 shows some distinct errors found by our tool in the benchmark. We were able to find all the data mapping errors in the benchmark. For example in file number 22,Listing 5.1 the map clause on line 29 is using the *alloc* attribute while line 34 is actually reading the array b, that was defined on line 18.

Table 5: Errors found in the DRACC Benchmark

Enum Type	Map Clause
DRACC File 22 Listing 5.1	ERROR Definition of :b on Line:18 is not reachable to Line:34, Missing Clause:to:Line:32
DRACC File 26 Listing 5.3	ERROR Definition of :c on Line:35 is not reachable to Line:46 Missing Clause:from/update:Line:44
DRACC File 30 Listing 5.4	ERROR Definition of :c on Line:25 is not reachable to Line:38 Missing Clause:to:Line:36
DRACC File 23 Listing 5.2	WARNING Line:30 maps partial data of :b smaller than its total size

Our analysis shows the following error for Listing 5.5, that was used as motivation in section 2,

```
ERROR Definition of :sum on Line:5 is not
reachable to Line:6 Missing Clause:from/update:Line:6
```

Since the user missed the explicit map clause for “sum”, the default mapping was first private, and the value was not mapped back to the host, hence line 6 used a stale version of the variable sum.

As explained in motivation section 2, for Listing 5.6 our tool is able to catch the data mapping issue, but please note that the Openmp 5.0 spec has been modified so that this is no longer a bug according to OpenMP 5.0.

```
ERROR Definition of :B on Line:8 is not
reachable to Line:12 Missing Clause:from/update:Line:10
```

Our tool is also able to catch the error for Listing 2.5, from section 2.

```
ERROR Definition of :A on Line:7 is not
reachable to Line:9 Missing Clause:from/update:8
```

The reason for this error was that, the reference count for variable A is incremented at line 5, after it was set to 1 at line 3, and device-host memory copy is inserted only if the reference count is decremented to zero.

Listing 5.5: Wrong Usage of default map on Reduction Scalar

```
1  int A[N], sum=0, i;
2  #pragma omp target
3  #pragma omp teams distribute parallel for
       reduction(+:sum)
4    for(i=0; i<N; i++)
5      sum += A[i];
6    printf("\n%d",sum);
```

Listing 5.6: Usage of alloc vs from

```
1  int A[10], B[10];
2  for (int i =0 ; i < 10 ; i++)
3      A[i] = i;
4
5  #pragma omp target enter data map(to:A
       [0:10]) map(alloc:B[0:10])
6  #pragma omp target map(alloc:B[0:10])
7  for (int i = 0 ; i < 10; i++)
8      B[i] = A[i];
9  #pragma omp target exit data map(from:B
       [0:10])
10
11 for (int i = 0 ; i < 10; i++)
12     printf("%d",B[i]);
```

6 Conclusion

References

1. AachenUniversity: Openmp benchmark https://github.com/RWTH-HPC/DRACC
2. LLVM: Llvm memoryssa https://llvm.org/docs/MemorySSA.html