OmpSan: OpenMP Code Sanitization, Static Verification of 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

OpenMP is a widely used directive-based parallel programming model, that supports offloading computations from hosts to device accelerators. Notable accelerator-related features in OpenMP 4.5 include unstructured data mapping, asynchronous execution, and runtime routines for device memory management.

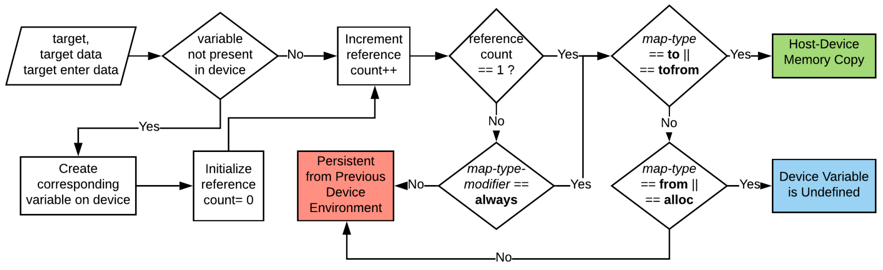
OMP 4.5 Target offloading and Data mapping OMP 4.5 offers the *omp target* directive for offloading computations to devices and the `omp target data` directive for mapping data across the host and the corresponding device data environment. On heterogeneous systems, managing the movement of data between the host and the device challenging, and is often a major source of performance and correctness bugs. In the OpenMP accelerator model, hosts and devices have their own memory space i.e., data environments and the data movement is supported either explicitly via the use of a `map` clause or, implicitly through default data-mapping rules. The optimal, or even correct specification of map clause is non-trivial and error-prone because it requires users to reason about the complex dataflow analysis: Users need to identify the statements and instructions that define variables at a given program point and follow the definition to all its uses in the program. Furthermore, following such data mapping behaviour for a given application is even more challenging. Given a data map construct, its semantics depends on all the previous usages of the map construct. Therefore, it is context sensitive and the entire call sequence leading up to the construct impacts its behavior.

1.1 OpenMP 4.5 Map Semantics

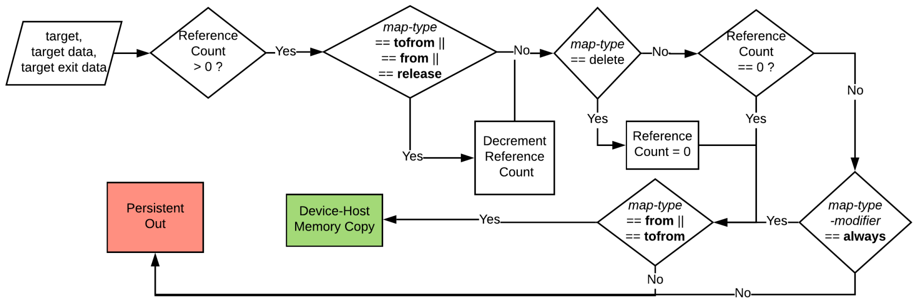
Figure 1 shows a schematic illustration of the complex set of rules used when mapping a host variable to the corresponding list item in the device data environment, as specified in the OpenMP 4.5 specification. In this work, we assume the device is a GPU, and mapping a variable from host to device introduces a host-device memory copy, and vice-versa. The different map types that OpenMP 4.5 supports are,

- alloc: allocate on device, uninitialized
- to: map to device before execution, (host-device memory copy)
- from: map from device after execution (device-host memory copy)
- tofrom:copy in and copy out the variable at the entry and exit of the device environment

The default map type for arrays is *tofrom*, while default for scalars is *firstprivate*, that is the only copy the value of the scalar at the entry to the device environment. The specification uses reference count of a variable, to decide when to


target, target data target enter data
variable not present in device
No
Yes
Increment reference count++
reference count == 1 ?
Yes
No
map-type == to || == tofrom
Yes
No
Host-Device Memory Copy
Create corresponding variable on device
Initialize reference count= 0
Persistent from Previous Device Environment
map-type-modifier == always
map-type == from || == alloc
Device Variable is Undefined

(a) Flowchart for Enter Device Environment


target, target data, target exit data
Reference Count > 0 ?
Yes
map-type == tofrom || == from || == release
No
Yes
Decrement Reference Count
map-type == delete
No
Yes
Reference Count = 0
Reference Count == 0 ?
No
Yes
map-type -modifier == always
map-type == from || == tofrom
Device-Host Memory Copy
Persistent Out

(b) Flowchart for Exit Device Environment

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

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 is incremented every time 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.

1.2 Our Solution

To address this issue as first-aid error detection, we propose a compile-time approach based on dataflow analysis. The key principle guiding our error detection approach is that "A user program yields the same result when enabling or disabling OpenMP constructs". 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.

- An algorithm to analyze the OpenMP runtime library calls inserted by clang in the LLVM IR, to infer the host/device memory copies
- A static analysis technique to validate if the host/device memory copies respect the original memory use-def relations.
- Reporting error/warnings on usage of data mapping constructs
- Diagnostic 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 issues and difficulties in usage of the *data map* construct. section 4 presents an overview of our approach to validate the usage of data mapping constructs. section 5 presents the LLVM implementation details, and section 6 presents the evaluation and some case studies.

2 Motivating Examples

We discuss potential errors in the user code arising from improper usage of the data mapping constructs, and illustrate how easy it is to incorrectly use the map construct. The accompanying examples motivate the utility and applicability of our proposed analysis and the tool OMPSan.

2.1 Default Scalar Mapping

Our first motivating example is, Listing 2.1. In this example, the definition of "sum" on line 5 does not reach line 6, since the "sum" does not have an explicit mapping and the default map for scalars is "firstprivate". As Listing 2.2 shows, an explicit map clause is essential to specify the copy in and copy out of 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);
```

2.2 Reference Count Issues

Example 1 Listing 2.3 shows an example of data-mapping attributes across different data environments. The array "B", is specified as "alloc" in the first data environment. According to OpenMP 4.5 (Figure 1) the reference count for this variable is set to 1 at line 5, as we enter a new data environment. Then we enter another 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, array "B" is not mapped back to host. By specifying the "from" attribute on the map clause for the variable "B" on line 9, the user would hope to see the updated value of B from the device to be available on the host as well. However, such an expectation would be incorrect, and easily missed by the user owing to reference-count rules that guide data copy-in/out outcomes. As such corresponding value of "B" on host is not updated. Listing 2.4 shows that replacing "alloc" with "from" on line 6, fixes this issue, and the programmer gets the expected behavior that is "B" is copied out at line 9.

Listing 2.3: Usage of alloc

```
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: Usage of 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(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]);
```

This behavior was actually changed in OpenMP 5.0, and even "alloc" would decrement the reference counter from 5.0.

This example shows the difficulty in interpreting an independent map construct. Especially when we are dealing with the global variables and map clauses across different functions, maybe even in different files, it becomes nearly impossible to understand and identify potential incorrect usages of the map construct. 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.

Example 2 Listing 2.5 shows another example, where because of reference count, user might not get the expected behavior. The line, 9 which executes on the host, does not read the value of "A" that was updated on device at line 7. This is again because the "from" clause on line 5, increments the reference count to 2 on entry, and back to 1 on exit, hence after line 7, "A" is not copied out to host. Listing 2.6 shows the usage of "update" to force the copy-out, and read the expected updated "A" on line 11.

Listing 2.5: Reference Count

```
define N 100
int A[N], sum=0;
#pragma omp target data map(from:A[0:N])
  {
    #pragma omp target map(from:A[0:N])
    for(int i=0; i<N; i++)
      A[i]=i;
    for(int i=0; i<N; i++)
      sum += A[i];
  }
```

Listing 2.6: Update Clause

```

define N 100
int A[N], sum=0;
#pragma omp target data map(from: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++)
      sum += A[i];
  }
```

In the next section, we will introduce our static analysis tool, which can identify such issues with the incorrect/unexpected usage of OpenMP data mapping constructs.

3 Background

3.1 Memory SSA

Our analysis is based on the LLVM Memory SSA [5] [7], which is an imprecise implementation of Array SSA[2]. Memory SSA captures the use-def chains for every memory access in the program. We construct the use-def chains for each array variable, based on the Memory SSA. Memory SSA is a virtual IR, with the following nodes,

- $INIT$, a special node to signify uninitialized or live on entry definitions
- $MemoryDef(N)$, corresponds to a memory store instruction, and where N is the definition that this definition node clobbers
- $MemoryUse(N)$, where N is the reaching definition, that this node uses
- $MemPhi(N_1, N_2, ...)$, where N_i are all the may reaching definitions

We make the following simplifying assumptions,

- Given an array variable we can find all the corresponding load and store instructions.
- A $MemoryDef$ node, clobbers the entire array associated with its store instruction.
- $MemoryPhi$ node are inserted only at the entry of basic blocks, which have more than one $MemoryDef$s that can flow into the basic block.
- We are concerned with only those array variables that are mapped to a target offload region.

3.2 Scalar Evolution Analysis

Scalar Evolution (SCEV) 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.

We use the SCEV analysis to symbolically evaluate the minimum and maximum value of every array index expression. In case the SCEV analysis fails to model an expression, we over approximate the range of the array indices to the maximum dimensions of the array variable.

section 5 has details of how we implement the analysis and handle different cases.

4 Our Approach

In this work, we assume that variables that have corresponding list items across host and device boundary or across different data environment boundaries on the device need to be updated at the edge of such boundaries. Typically this assumption reflects the practical use-case of the variables. For example, in Listing 2.3, a user would expect the updated value of "B" after the second data environment on line 12. Having said that, a skilled ninja programmer may very well expect "B" to remain stale, because of his knowledge and understanding of the complexities of data mapping rules. Our analysis and error/warning reports from this work are intended only for the former case.

Here we first outline the key steps of our approach on algorithm and exemplify it with two concrete examples to illustrate the algorithm in action.

4.1 Algorithm

The Algorithm 1 shows an overview of the algorithm for the data map analysis. Firstly we inline all user defined functions, and then use the Memory SSA to construct the memory use-def chains, for every array variable. After step 2, we get a graph for each memory array, with an edge from a $MemoryDef$ or $MemPhi$ to the corresponding $MemoryUse$ or $MemPhi$ for the reaching definition relation. Then we use the Scalar Evolution Analysis, to compute the range of locations accessed by every memory reference. If the analysis fails to compute the range of locations statically, then we over-approximate it to the maximum size of the array.

Now we interpret the semantics of the `target` construct, to classify the Memory SSA nodes, as executing on host or device. Since the device and host have a separate data environment, they refer to different array variables. So in step 5, we remove the edges between host and device nodes. Now according to the semantics of the `map` clause, the host variables are mapped to the device variables, and vice-versa. At any program point, when a host variable is mapped to device, we find its corresponding reaching definition, and add a special edge from the host $MemoryDef$ or $MemPhi$, to the $MemoryUse$ or $MemPhi$ on the device. Similarly when mapping a device variable to host variable, we add the edge between the corresponding definition on the device to the corresponding use on the host. In step 6, we also annotate the edges with the array sections according to the `map` clause.

Algorithm 1 Overview of Data Mapping Analysis

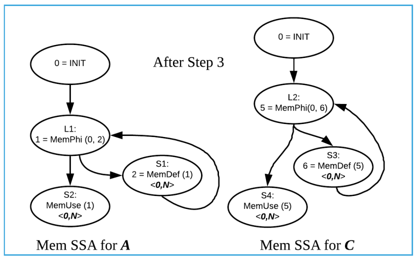
```
1: Construct the Memory SSA for each array variable
2: Do scalar evolution analysis to compute the range of each memory access
3: Interpret the omp target constructs and mark the nodes in the SSA graphs as host
   and device
4: Remove the edges from the Memory SSA graph which connect device nodes with
   host nodes
5: Interpret map clauses to insert edges in the Memory SSA graph, that correspond
   to host-device and device-host memory copies, annotated with array sections
6: For every node in Memory SSA graph, if the edge from reaching definition is miss-
   ing, then Report ERROR
7: For every edge connecting host, device nodes, the array section of the edge must
   be a subset of the memory access range of the destination node, else report it as
   Warning
```

Finally, we report an error for every missing use-def edge in our graph, since it indicates a mismatch between the sequential use-def relations and the OpenMp reaching definitions. We report a warning, when an edge indicates that the array is partially mapped, and the $MemUse$ could potentially access locations outside the mapped range.

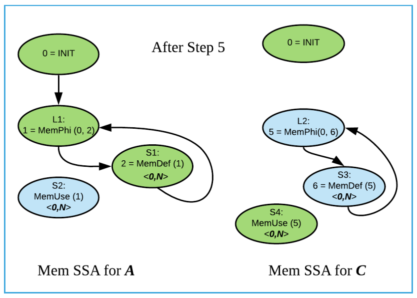
```

L1: for ( i =0 ; i < 100 ; i++) {
          S1: A[i] = 0;
    }
#pragma omp target enter data map(to:A[0:N],
      alloc:C[0:N])
#pragma omp target
L2: for (i=0; i < 100; i++ ) {
          S2: t = A[i] ;
          S3 C[i] = t;
    }
#pragma omp target exit data map( release:C[0:N])
L3: for (i=0; i < 100; i++) {
          S4: print(C[i])
    }
```

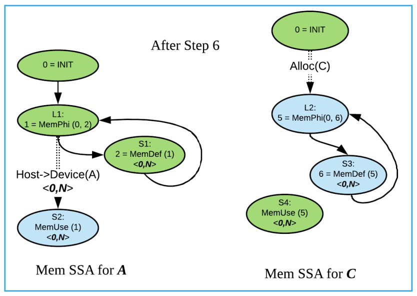
(a) Example 1, user Code



(b) Memory SSA



(c) Classify Host/Device Regions



(d) Host/Device Memory Copies

Fig. 2: Data Map Analysis Example 1

Example 1 Let us consider the first example Figure 2a to illustrate our approach for analysis of data mapping clauses. As per Algorithm 1, Figure 2b shows the memory SSA for arrays “A” and “C”. Figure 2c shows the resulting graph after steps 5,6. Figure 2d connects $L1$ with $S2$ with a host-device memory copy for the enter data map pragma with `to`: $A[o : N]$ on line 5. Also, we connect $INIT$ node with $L2$, to account for the `alloc`:$C[0 : N]$, which means an uninitialized reaching definition. Now, we can conclude our analysis with the following report

– Error, Node $S4$:$MemUse(5)$ is not connected with its reaching definition from $L2 : 5 = MemPhi(0, 6)$

Example 2 For our next example, We modify the Figure 2a, 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])
```

Figure 3 shows the Memory SSA graphs after step 3, and the final graph after step 6. We report the following warning, after our analysis,

– Warning, Node $S2$:$MemUse(1)$ uses, $< 0, 100 >$, which is not a subset of $< 0, 50 >$

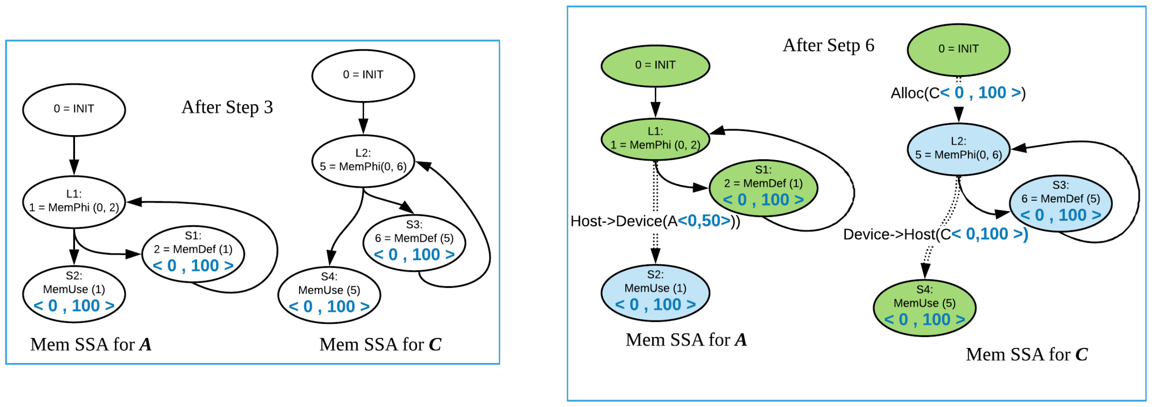


Fig. 3: Data Map Analysis Example 2

5 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, its difficult to perform a global data flow analysis for the memory use-def information of the offloaded region. To simplify the analysis, 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 5.1. During the second pass, we perform the whole program context and flow sensitive data flow analysis, to construct the Memory use-def chains, subsection 5.2. Then this pass validates if the "target map" information generated by the previous pass, respects all the Memory use def relations.

5.1 Interpreting OpenMP pragmas

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.

Listing 5.1: Example map clause

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

Listing 5.2: 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 5.1 shows a very simple user program, with the target data map clause. Listing 5.2 shows the corresponding LLVM IR in pseudocode, 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 5.2 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,

Table 1: Target Runtime Library Routines

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	`int64_t device_id, int32_t num_args void** args_base, void** args, int64_t *args_size, int64_t *args_maptype`
__tgt_target_data_update :: Make a set of values consistent between host and device	`int64_t device_id, int32_t num_args void** args_base, void** args, int64_t *args_size, int64_t *args_maptype`
__tgt_target :: Begin data environment, launch target region execution and end device environment	`int64_t device_id, void *host_addr, int32_t num_args void** args_base, void** args, int64_t *args_size, int64_t *args_maptype`
__tgt_target_teams :: Same as above, also specify number of teams and threads	`int64_t device_id, void *host_addr, int32_t num_args, void** args_base, void** args, int64_t *args_size, int64_t *args_maptype, int32_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

– CopyIn: A memory copy is introduced from the host to the corresponding list item in 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 in Listing 5.3. Table 4 shows the data mapping inferred by our tool. For example "B" is persistent out of the first target region, that ends on line 9, and persistent in to the second target region on line 13. "B" is copy out only at the `exit data map` on line 13.

Listing 5.3: 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(from: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 data map(from:B[0:10],C[0:N])
11    for ( int i = 0 ; i < 10; i ++)
12      C [ i ] = B [ i ]*i;
13  #pragma omp target exit data map(from:B[0:10])
14    for (int i = 0 ; i < 10; i++)
15      printf("%d",B[i]);
16
17      return 0;
18  }
```

Table 4: Output Data mapping for Listing 5.3

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	9		A[0:10], B[0:10]	A[0:10]	B[0:10]	
__tgt_target	10	12	A[0:10]	B[0:10]	A[0:10]	B[0:10]	
__tgt_target_data_end	13	13			B[0:10]		

5.2 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. We also perform inlining of all user functions to enable 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[5]. 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. Listing 5.4 shows an example, with Figure 4 as the simplified MemorySSA.

Listing 5.4: Example of MemorySSA

```
 1  int main(){
 2    int A[10], B[10];
 3    for (int i =0 ; i < 10 ; i++) {
 4      // %arrayidx = getelementptr %A, 0, %idxprom
 5      // store %i.0, %arrayidx ,
 6      A[i] = i;
 7    }
 8  #pragma omp target enter data map(to:A[0:5])
 9          map(alloc:B[0:10])
10  #pragma omp target
11    for (int i = 0 ; i < 10; i++) {
12      // %arrayidx7 = getelementptr %A, 0, %idxprom6
13      // %2 = load %arrayidx7
14      int t = A[i];
15      // %arrayidx9 = getelementptr %B, 0, %idxprom8
16      // store %2, %arrayidx9
17      B[i] = t
18    }
19
20    for (int i = 0 ; i < 10; i++) {
21      //arrayidx19 = getelementptr %B, 0, %idxprom18
22      //%3 = load %arrayidx19
23      printf("%d",B[i]);
24    }
25
26      return 0;
27  }
```

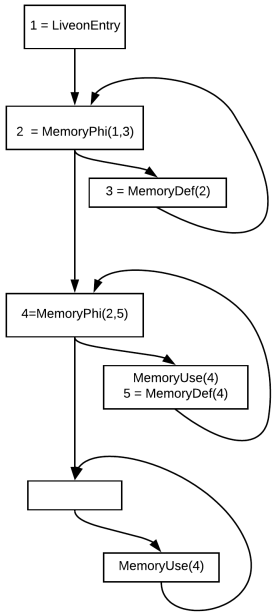


Fig. 4: MemorySSA

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.

Now, given this MemorySSA, we can extract the Array use-def chains, if we can disambiguate the memory variable that the load/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 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.

5.3 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 5.1 interprets the memory copies introduced by the omp map clauses. Now as explained in section section 4, 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.

5.4 Scalar Evolution Analysis, Array Sections

For our analysis, we need the range of a 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 section 4 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.

6 Evaluation and Case Studies

Listing 6.1: DRACC File 22

```c
int init(){
    for(int i=0; i<C; i++){
        for(int j=0; j<C; j++){
            b[j+i*C]=1;
        }
        a[i]=1;
        c[i]=0;
    }
        return 0;
}


int Mult(){

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

Listing 6.2: DRACC File 26

```c
    #pragma omp target
              enter data map(to:a[0:C],b[0:C*C
                   ],c[0:C]) device(0)
    #pragma omp target device(0)
    {
        #pragma omp teams
                  distribute parallel for
        for(int i=0; i<C; i++){
            for(int j=0; j<C; j++){
                c[i]+=b[j+i*C]*a[j];
            }
        }
    }
    #pragma omp target exit
              data map(release:c[0:C])
              map(release:a[0:C],b[0:C*C])
                   device(0)
    return 0;
}

int check(){
    bool test = false;
    for(int i=0; i<C; i++){
        if(c[i]!=C){
```

Listing 6.3: DRACC File 23

```c
int Mult(){

     #pragma omp target map(to:a[0:C],b[0:C])
    map(tofrom:c[0:C]) device(0)
     {
         #pragma omp teams
                   distribute parallel for
         for(int i=0; i<C; i++){
             for(int j=0; j<C; j++){
                 c[i]+=b[j+i*C]*a[j];
             }
         }
     }
```

Listing 6.4: DRACC File 30

```c
int init(){
    for(int i=0; i<C; i++){
        for(int j=0; j<C; j++){
            b[j+i*C]=1;
        }
        a[i]=1;
        c[i]=0;
    }
~
int Mult(){

    #pragma omp target
        map(to:a[0:C],b[0:C*C]) map(from:c[0:
              C*C]) device(0)
    {
        #pragma omp teams
                  distribute parallel for
        for(int i=0; i<C; i++){
            for(int j=0; j<C; j++){
                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 6.1 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, so our tool points out the error message that the "to" clause is mssing. For Listing 6.2 "c" is updated on the device at line 34, but it is not mapped back, and our error points out that the "from/update" clause is missing. For Listing 6.4, "c" is initialized on line 25, but the host variable is not mapped to the device, when creating the device environment at line 33. For Listing 6.3, only "C" elements of "b" are mapped to the device, while the actual size of "b" is "C*C". Our tool is able to track the malloc call to flag this usage as a warning.

Table 5: Errors found in the DRACC Benchmark

File Name	Error/Warning
DRACC File 22 Listing 6.1	ERROR Definition of :b on Line:18 is not reachable to Line:34, Missing Clause:to:Line:32
DRACC File 26 Listing 6.2	ERROR Definition of :c on Line:35 is not reachable to Line:46 Missing Clause:from/update:Line:44
DRACC File 30 Listing 6.4	ERROR Definition of :c on Line:25 is not reachable to Line:38 Missing Clause:to:Line:36
DRACC File 23 Listing 6.3	WARNING Line:30 maps partial data of :b smaller than its total size
Motivation Ex Listing 2.1	ERROR Definition of :sum on Line:5 is not reachable to Line:6 Missing Clause:from/update:Line:6
Motivation Ex Listing 2.3	ERROR Definition of :B on Line:8 is not reachable to Line:12 Missing Clause:from/update:Line:10
Motivation Ex Listing 2.5	ERROR Definition of :A on Line:7 is not reachable to Line:9 Missing Clause:from/update:8

The last 3 rows of the table shows the errors pointed out by our tool, for the examples from the motivation section. The errors were explained in section 2

6.1 Time to Compile

Table 6 shows the time to run OmpSan, on few SPEC ACCEL and NAS parallel benchmarks. We compare it with the time to compile the programs with "-O3" flag. Due to the context and flow sensitive data flow analysis implemented in OmpSan, its runtime is significant, and almost as much as the time to compile the entire program.

6.2 Diagnostic Information

Another major use case for our tool, is to help understand the data mapping behavior of an existing source code. For example, Listing 6.5 shows a code fragment from the benchmark "FT" of "NAS" suite. We argue that it is very difficult

Table 6: Time to Run OmpSan

Benchmark Name	-O3 Compilation Time (sec)	OmpSan Runtime (sec)
SPEC 504.polbm	17	16
SPEC 503.postencil	3	3
SPEC 552.pep	7	4
SPEC 554.pcg	15	9
NAS FT	32	15
NAS MG	34	31

to read this openmp target code, as line 311 explicitly maps every array variable with the "alloc" map type. Since, these are global variables, the semantics of this `target map` depend on the previous occurrences of the mapping clause. Our tool can generate the following information, which shows the current state of the data mapping clause.

- *__tgt_target_teams*, from::"ft.c:311" to "ft.c:331"
- Alloc: $u0\_imag[0:8421376], u0\_real[0:8421376]$
- Persistent In :: $twiddle[0:8421376], u1\_imag[0:8421376], u1\_real[0:8421376]$
- Persistent Out :: $twiddle[0:8421376], u0\_imag[0:8421376], u0\_real[0:8421376], u1\_imag[0:8421376], u1\_real[0:8421376]$
- Copy In:: *Null*, Copy Out:: *Null*

Similarly, the data mapping for the function $cffts3\_pos$ Listing 6.6, is as shown below,

- *__tgt_target_teams*, from::"ft.c:1180" to "ft.c:1276"
- Persistent In :: $gty1\_imag[0:16777216], gty1\_real[0:16777216], gty2\_imag[0:16777216], gty2\_real[0:16777216], logd3[0:0], u0\_imag[0:8421376], u0\_real[0:8421376], u1\_imag[0:8421376], u1\_real[0:8421376], u\_imag[0:257], u\_real[0:257]$
- Persistent Out: $u1\_imag[0:8421376], u1\_real[0:8421376], u\_imag[0:257], u\_real[0:257]$
- Copy Out: $gty1\_imag[0:16777216], gty1\_real[0:16777216], gty2\_imag[0:16777216], gty2\_real[0:16777216], u0\_imag[0:8421376], u0\_real[0:8421376]$

Listing 6.5: *evolve* from NAS/ft.c

```
static void evolve(int d1, int d2, int d3)
{
  int i, j, k;
// ...
#pragma omp target map (alloc: u0_real,
  u0_imag,u1_real,u1_imag,twiddle)
  {
#pragma omp teams distribute
    for (k = 0; k < d3; k++) {
#pragma omp parallel for
      for (j = 0; j < d2; j++) {
#pragma omp simd
        for (i = 0; i < d1; i++) {
          u0_real[k*d2*(d1+1) + j*(d1+1) + i
  ] = u0_real[k*d2*(d1+1) + j*(d1+1) + i]

  *twiddle[k*d2*(d1+1) + j*(d1+1) + i];
          u0_imag[k*d2*(d1+1) + j*(d1+1) + i
  ] = u0_imag[k*d2*(d1+1) + j*(d1+1) + i]

  *twiddle[k*d2*(d1+1) + j*(d1+1) + i];
```

Listing 6.6: *cffts3_pos* from NAS/ft.c

```
static void cffts3_pos(int is, int d1, int
  d2, int d3)
{
  int logd3;
  int i, j, k, ii;
  int l, j1, i1, k1;
  int n1, li, lj, lk, ku, i11, i12, i21, i22
  ;
  double uu1_real, x11_real, x21_real;
  double uu1_imag, x11_imag, x21_imag;
  double uu2_real, x12_real, x22_real;
  double uu2_imag, x12_imag, x22_imag;
  double temp_real, temp2_real;
  double temp_imag, temp2_imag;

  logd3 = ilog2(d3);
~
#pragma omp target teams distribute collapse
  (2)
    for (j = 0; j < d2; j++) {
//#pragma acc loop vector independent
//#pragma omp simd
      for (i = 0; i < d1; i ++) {
#pragma omp parallel for
        for (k = 0; k < d3; k++) {
          gty1_real[j][k][i] = u1_real[k*d2
  *(d1+1) + j*(d1+1) + i];
          gty1_imag[j][k][i] = u1_imag[k*d2
  *(d1+1) + j*(d1+1) + i];
        }

```

7 Related Work and Conclusion

Managing data transfers to and from GPUs has always been an important problem for GPU programming. Several solutions have been proposed to help the programmer in managing the data movement. The OpenMPC compiler [4] also proposed a static analysis to insert data transfers automatically. Seyong et. al proposed a directive based approach, that combined compile-time/runtime method to verify the correctness of CPU-GPU memory transfer and even optimize it in [3]. Pai et. al [8] proposed a compiler analysis to detect potential stale accesses and uses a runtime to initiate transfers as necessary, for the X10 compiler. [9] also proposed a static analysis technique to optimize the data transfers for the X10. [6] has also worked on automatically inferring the OpenMP mapping clauses using some static analysis.

In this paper, we have developed a static analysis technique to interpret the semantics of the openmp map clause, and deduce the data transfers introduced by the clause.

References

1. AachenUniversity: Openmp benchmark https://github.com/RWTH-HPC/DRACC
2. Knobe, K., Sarkar, V.: Array ssa form and its use in parallelization. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 107–120. POPL '98, ACM, New York, NY, USA (1998). https://doi.org/10.1145/268946.268956, http://doi.acm.org/10.1145/268946.268956

3. Lee, S., Li, D., Vetter, J.S.: Interactive program debugging and optimization for directive-based, efficient gpu computing. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 481–490 (May 2014). https://doi.org/10.1109/IPDPS.2014.57
4. Lee, S., Eigenmann, R.: Openmpc: Extended openmp programming and tuning for gpus. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. SC '10, IEEE Computer Society, Washington, DC, USA (2010). https://doi.org/10.1109/SC.2010.36, https://doi.org/10.1109/SC.2010.36
5. LLVM: Llvm memoryssa https://llvm.org/docs/MemorySSA.html
6. Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., Pereira, F.M.Q.a.: Dawncc: Automatic annotation for data parallelism and offloading. ACM Trans. Archit. Code Optim. **14**(2), 13:1–13:25 (May 2017). https://doi.org/10.1145/3084540, http://doi.acm.org/10.1145/3084540
7. Novillo, D.: Memory ssa- a unified approach for sparsely representing memory operations. In: Proceedings of the GCC Developers' Summit (2007)
8. Pai, S., Govindarajan, R., Thazhuthaveetil, M.J.: Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. pp. 33–42. PACT '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2370816.2370824, http://doi.acm.org/10.1145/2370816.2370824
9. Thangamani, A., Nandivada, V.K.: Optimizing remote data transfers in x10. In: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques. pp. 27:1–27:15. PACT '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3243176.3243209, http://doi.acm.org/10.1145/3243176.3243209