OMPSan: Static Verification of OpenMP's 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, Atlanta, Georgia
prithayan@gatech.edu
[2] IBM Toronto Laboratory, Markham, Canada

Abstract. OpenMP offers directives for offloading computations from CPU hosts to accelerator devices such as GPUs. A key underlying challenge is in efficiently managing the movement of data across the host and the accelerator. User experiences have shown that memory management in OpenMP programs with offloading capabilities is non-trivial and error-prone.

This paper presents **OMPSan** (OpenMP Sanitizer) – a static analysis-based tool that helps developers detect bugs from incorrect usage of the `map` clause, and also suggests potential fixes for the bugs. We have developed an LLVM based data flow analysis that validates if the def-use information of the array variables are respected by the mapping constructs in the OpenMP program. We evaluate **OmpSan** over some standard benchmarks and also show its effectiveness by detecting commonly reported bugs.

Keywords: OpenMP offloading · OpenMP target data mapping · LLVM · Memory management · Static analysis · Verification · Debugging

1 Introduction

Open Multi-Processing (OpenMP) is a widely used directive-based parallel programming model that supports offloading computations from hosts to accelerator devices such as GPUs. Notable accelerator-related features in OpenMP include unstructured data mapping, asynchronous execution, and runtime routines for device memory management.

OMP Target Offloading and Data Mapping. OMP 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 can be challenging, and is often a major source of performance and correctness bugs. In the OpenMP accelerator model, data movement between device and host is supported either explicitly via the use of a `map` clause

X. Fan et al. (Eds.): IWOMP 2019, LNCS 11718, pp. 3–18, 2019.
https://doi.org/10.1007/978-3-030-28596-8_1

or, implicitly through default data-mapping rules. The optimal, or even correct, specification of map clauses can be non-trivial and error-prone because it requires users to reason about the complex dataflow analysis. To ensure that the map clauses are correct, the OpenMP programmers need to make sure that variables that are defined in one data environments and used in another data environments are mapped accordingly across the different device and host data environments. Given a data map construct, its semantics depends on all the previous usages of the map construct. Therefore, dataflow analysis of map clauses is necessarily context-sensitive since the entire call sequence leading up to a specific map construct can impact its behavior.

1.1 OpenMP 5.0 Map Semantics

Figure 1 shows a schematic illustration of the set of rules used when mapping a host variable to the corresponding list item in the device data environment, as specified in the OpenMP 5.0 standard. The rest of this paper assumes that the accelerator device is a GPU, and that mapping a variable from host to device introduces a host-to-device memory copy, and vice-versa. However, the bugs that we identify reflect errors in the OpenMP code regardless of the target device.

The different map types that OpenMP 5.0 supports are,

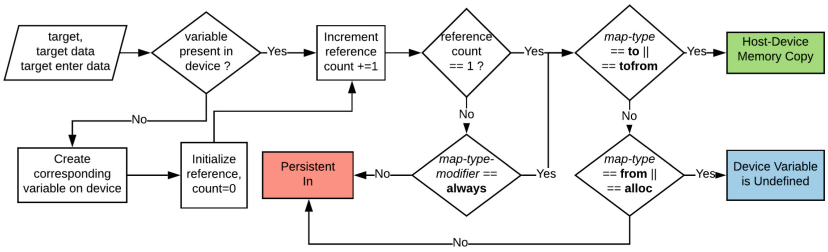
- `alloc`: allocate on device, uninitialized
- `to`: map to device before kernel execution, (host-device memory copy)
- `from`: map from device after kernel execution (device-host memory copy)
- `tofrom`: copy in and copy out the variable at the entry and exit of the device environment.

Arrays are implicitly mapped as `tofrom`, while scalars are firstprivate in the target region implicitly, *i.e.,* the value of the scalar on the host is copied to the corresponding item on the device only at the entry to the device environment. As Fig. 1 shows, OpenMP 5.0 specification uses the reference count of 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 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 mapped back to the host from the device.

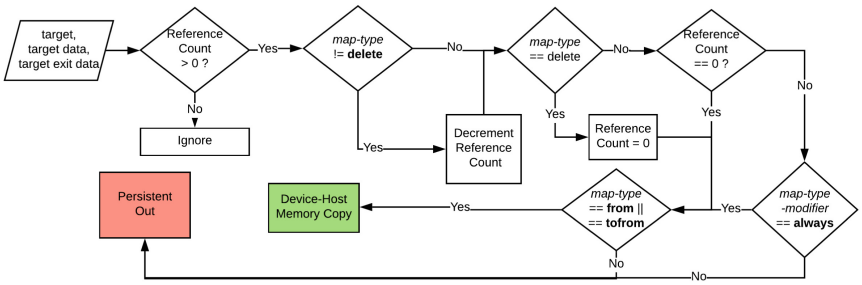
1.2 The Problem

For target offloading, the map clause is used to map variables from a task's data environment to the corresponding variable in the device data environment. Incorrect data map clauses can result in usage of stale data in either host or device data environment, which may result in the following kinds of issues,

- When reading the variable on the device data environment, it does not contain the updated value of its original variable.
- When reading the original variable, it was not updated with the latest value of the corresponding device environment variable.



(a) Flowchart for Enter Device Environment



(b) Flowchart for Exit Device Environment

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

1.3 Our Solution

We propose a static analysis tool called OMPSan to perform OpenMP code "sanitization". OMPSan is a compile-time tool, which statically verifies the correctness of the data mapping constructs based on a dataflow analysis. The key principle guiding our approach is that: *an OpenMP program is expected to yield the same result when enabling or disabling OpenMP constructs*. Our approach detects errors by comparing the dataflow information (reaching definitions via LLVM's memory SSA representation [10]) between the OpenMP and baseline code. We developed an LLVM-based implementation of our approach and evaluated its effectiveness using several case studies. Our specific contributions include:

- an algorithm to analyze OpenMP runtime library calls inserted by Clang in the LLVM IR, to infer the host/device memory copies. We expect that this algorithm will have applications beyond our OMPSan tool.
- a dataflow analysis to infer Memory def-use relations.
- a static analysis technique to validate if the host/device memory copies respect the original memory def-use relations.
- diagnostic information for users to understand how the map clause affects the host and device data environment.

Even though our algorithm is based on clang OpenMP implementation, it can very easily be applied to other approaches like using directives to delay the OpenMP lowering to a later LLVM pass. The paper is organized as follows. Section 2 provides motivating examples to describe the common issues and difficulties in using OpenMP's *data map* construct. Section 3 provides the background information that we use in our analysis. Section 4 presents an overview of our approach to validate the usage of data mapping constructs. Section 5 presents the LLVM implementation details, and Sect. 6 presents the evaluation and some case studies. Subsection 6.3 also lists some of the limitations of our tool, some of them common to any static analysis.

2 Motivating Examples

To motivate the utility and applicability of OMPSan, we discuss three potential errors in user code arising from improper usage of the data mapping constructs.

2.1 Default Scalar Mapping

Example 1: Consider the snippet of code in Listing 2.1. The `printf` on host, line 8, prints stale value of sum. Note that the definition of `sum` on line 5 does not reach line 8, since the variable `sum` is not mapped explicitly using the `map` clause. As such, `sum` is implicitly `firstprivate`. As Listing 2.2 shows, an explicit `map` clause with the `tofrom` attribute is essential to specify the copy in and copy out of `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     }
7   }
8   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     }
7   }
8   printf("\n%d",sum);
```

2.2 Reference Count Issues

Example 2: Listing 2.3 shows an example of a reference count issue. The statement in line 12, which executes on the host, does not read the updated value of A from the device. This is again because of the `from` clause on line 5, which increments the reference count to 2 on entry, and back to 1 on exit, hence after line 10, A is not copied out to host. Listing 2.4 shows the usage of `target update` directive to force the copy-out and to read the updated value of A on line 15.

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 difficult to understand and identify potential incorrect usages of the map construct.

Listing 2.3. Reference Count

```
1  #define N 100
2  int A[N], sum=0;
3  #pragma omp target data
4      map(from:A[0:N]) {
5      #pragma omp target
6          map(from:A[0:N]) {
7      for(int i=0; i<N; i++) {
8          A[i]=i;
9        }
10     }
11     for(int i=0; i<N; i++) {
12       sum += A[i];
13     }
14   }
```

Listing 2.4. Update Clause

```
1
2  #define N 100
3  int A[N], sum=0;
4  #pragma omp target data
5              map(from:A[0:N]) {
6      #pragma omp target
7        map(from:A[0:N]) {
8        for(int i=0; i<N; i++) {
9          A[i]=i;
10       }
11     }
12     #pragma omp target
13         update from(A[0:N])
14     for(int i=0; i<N; i++) {
15       sum += A[iGhosh];
16     }
17   }
```

3 Background

OMPSan assumes certain practical use cases, for example, in Listing 2.3, a user would expect the updated value of A on line 12. Having said that, a skilled ninja programmer may very well expect A to remain stale, because of their knowledge and understanding of the complexities of data mapping rules. Our analysis and error/warning reports from this work are intended primarily for the former case.

3.1 Memory SSA Form

Our analysis is based on the LLVM Memory SSA [10,12], which is an imprecise implementation of Array SSA [7]. The Memory SSA is a virtual IR, that captures the def-use information for array variables. Every definition is identified by a unique name/number, which is then referenced by the corresponding use.

The Memory SSA IR has the following kinds of instructions/nodes,

- $INIT$, a special node to signify uninitialized or live on entry definitions
- $N' = MemoryDef(N)$, N' is an operation which may modify memory, and N identifies the last write that N' clobbers.
- $MemoryUse(N)$, is an operation that uses the memory written by the definition N, and does not modify the memory.
- $MemPhi(N_1, N_2, ...)$, is an operation associated with a basic block, and N_i is one of the may reaching definitions, that could flow into the basic block.

We make the following simplifying assumptions, to keep the analysis tractable

- Given an array variable we can find all the corresponding load and store instructions. So, we cannot handle cases, when pointer analysis fails to disambiguate the memory a pointer refers to.

- A $MemoryDef$ node clobbers the array associated with its store instruction. As a result, write to any array location, is considered to update the entire array.
- We analyze only the array variables that are mapped to a target region.

3.2 Scalar Evolution Analysis

LLVM's 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 index expressions of the memory operations.

We implemented an analysis for array sections, that given a load/store, uses the LLVM SCEV analysis, to compute the minimum and maximum values of the corresponding index into the memory access. If the analysis fails, then we default to the maximum array size, which is either a static array, or can be extracted from the LLVM memory *alloc* instructions.

4 Our Approach

In this section, we outline the key steps of our approach with the algorithm and show a concrete example to illustrate the algorithm in action.

4.1 Algorithm

Algorithm 1 shows an overview of our data map analysis algorithm. First, we collect all the array variables used in all the map clauses in the entire module. Then line 5, calls the function `ConstructArraySSA`, which constructs the Array SSA for each of the mapped Array variables. (In this paper, we use "Array SSA" to refer to our extensions to LLVM's Memory SSA form by leveraging the capabilities of Array SSA form [7].) Then, we call the function, `InterpretTargetClauses`, which modifies the Array SSA graph, in accordance of the map semantics of the program. Then finally `ValidateDataMap` checks the reachability on the final graph, to validate the map clauses, and generates a diagnostic report with the warnings and errors.

Example. Let us consider the example in Fig. 2a to illustrate our approach for analysis of data mapping clauses. `ConstructArraySSA` of Algorithm 1, constructs the memory SSA form for arrays "A" and "C" as shown in Fig. 2b. Then, `InterpretTargetClauses`, removes the edges between host and device nodes, as shown in Fig. 2c, where the host is colored green and device is blue. Finally, the loop at line 29 of the function `InterpretTargetClauses`, introduces the host-device/device-host memory copy edges, as shown in Fig. 2d. For example $L1$ is connected to $S2$ with a host-device memory copy for the enter data map pragma

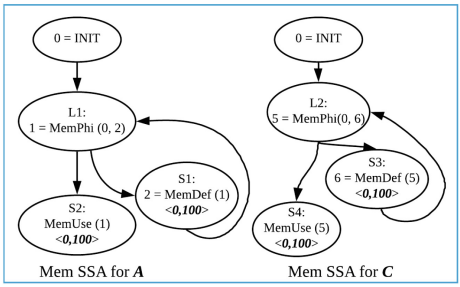
Algorithm 1. Overview of Data Mapping Analysis

```
1: function DataMapAnalysis(Module)
2:     MappedArrayVars = φ
3:     for ArrayVar ∈ MapClauses do
4:         MappedArrayVars = MappedArrayVars ∪ ArrayVar
5:     ConstructArraySSA(Module, MappedArrayVars)
6:     InterpretTargetClauses(Module, MappedArrayVars)
7:     ValidateDataMap(MappedArrayVars)
8: function ConstructArraySSA(Module, MappedArrayVars)
9:     for MemoryAccess ∈ Module do
10:        ArrayVar = getArrayVar(MemoryAccess)
11:        if ArrayVar ∈ MappedArrayVars then
12:            if MemoryAccess ∈ OMP_targetOffload_Region then
13:                targetNode = true                    ▷ If Memory Access on device
14:            else
15:                targetNode = false                   ▷ If Memory Access on host
16:            Range = SCEVGetMinMax(MemoryAccess)
17:            underConstruction= GetArraySSA(ArrayVar)
18:                                                     ▷ could be null or incomplete
19:            InsertNodeArraySSA(underConstruction,MemoryAccess, targetNode, Range
   )
20:                                                     ▷ Incrementally construct, by adding this access
21: function InterpretTargetClauses(Module, MappedArrayVars)
22:    for ArrayVar ∈ MappedArrayVars do
23:        ArraySSA = GetArraySSA(ArrayVar)
24:        for edge, (node, Successornode) ∈ (ArraySSA) do
25:            nodeIsTarget = isTargetOffload(node)
26:            succIsTarget = isTargetOffload(Successornode)
27:            if nodeIsTarget != succIsTarget then
28:                RemoveArraySSAEdge(node, Successornode )
29:    for dataMap ∈ dataMapClauses do
30:        hostNode = getHostNode(dataMap)
31:        deviceNode = getDeviceNode(dataMap)
32:        mapType = getMapClauseType(dataMap)
33:                                                    ▷ alloc/copyIn/copyOut/persistentIn/persistentOut
34:        InsertDataMapEdge(hostNode, deviceNode, mapType)
35: function ValidateDataMap(MappedArrayVars)
36:    for ArrayVar ∈ MappedArrayVars do
37:        ArraySSA = GetArraySSA(ArrayVar)
38:        for memUse ∈ getMemoryUseNodes(ArraySSA) do
39:            useRange = getReadRange(memUse)
40:            clobberingAccess = getClobberingAccess(ArraySSA, memUse)
41:            if isPartiallyReachable(ArraySSA, clobberingAccess, memUse, useRange)
   then
42:                Report WARNING
43:            else if isNotReachable(ArraySSA, clobberingAccess, memUse) then
44:                Report ERROR
```

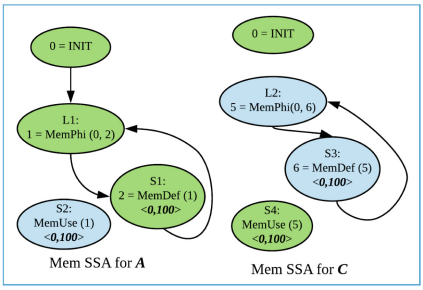
with `to`: $A[0:50]$ on line 5. Also, we connect the *INIT* node with $L2$, to account for the `alloc`:$C[0:100]$, which implies an uninitialized reaching definition for this example.

```
1
2  L1: for ( i =0 ; i < 100 ; i++) {
3          S1: A[i] = 0;
4      }
5  #pragma omp target enter data map(
       to:A[0:50], alloc:C[0:100])
6  #pragma omp target
7  L2: for (i=0; i < 100; i++ ) {
8          S2: t = A[i] ;
9          S3 C[i] = t;
10     }
11 #pragma omp target exit data map(
       release:C[0:100])
12 L3: for (i=0; i < 100; i++) {
13         S4: print(C[i])
14     }
```

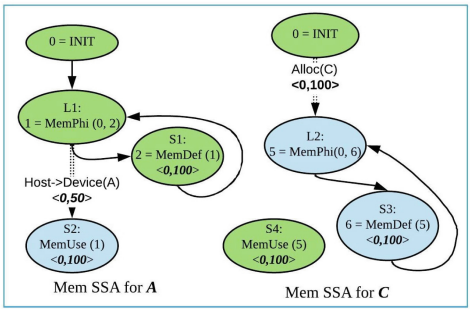
(a) Example, user Code



(b) Memory SSA for sequential version



(c) Classify Host/Device Regions (green=host, blue=device)



(d) Host/Device Memory Copies

Fig. 2. Example of data map analysis

Lastly, `ValidateDataMap` function, traverses the graph, resulting in the following observations:

- (Error) Node $S4{:}MemUse(5)$ is not reachable from its corresponding definition $L2 : 5 = MemPhi(0, 6)$
- (Warning) Only the partial artial array section $A[0 : 50]$, is reachable from definition $L1 : 1 = MemPhi(0, 2)$ to $S2 : MemUse(1)\langle 0 : 100\rangle$

Section 6 contains other examples of the errors and warnings discovered by our tool.

5 Implementation

We implemented our framework in LLVM 8.0.0. 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 limitations of this approach with respect to high level analysis like the one OMPSan is trying to accomplish. For example, the region of code that needs to be offloaded to a device is opaque since it is moved to a separate function. These functions are in turn called from the

OpenMP runtime library. As a result, it is challenging to perform a global data flow analysis for the memory def-use information of the offloaded region. To simplify the analysis, we have to compile with clang twice.

First, we compile the OpenMP program with the flag that enables parsing the OpenMP constructs, and compile it again without the flag, so that Clang ignores the OpenMP constructs and instead generates the baseline LLVM IR for the sequential version. During the OpenMP compilation pass, we execute our analysis pass, which parses the runtime library calls and generates a csv file that records all the user specified "target map" clauses, as explained in Subsect. 5.1.

Next we compile the program by ignoring the OpenMP pragmas, and perform whole program context and flow sensitive data flow analysis on LLVM code generated from the sequential version, to construct the Memory def-use chains, explained in Subsect. 5.2. Then this pass validates if the "target map" information recorded in the csv file, respects all the Memory def-use relations present in the sequential version of the code.

5.1 Interpreting OpenMP Pragmas

Listing 5.1. Example map clause

```
1
2  #pragma omp target
3      map(tofrom:A[0:10])
4      for (i = 0 ; i < 10; i++)
5        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 = {40}
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 a target data map clause. Listing 5.2 shows the corresponding LLVM IR in pseudocode, after clang introduces the runtime calls at Line 5. We parse the arguments of this call to interpret the map construct. For example, the 3rd argument to the call at line 6 of Listing 5.2 is 1, that means there is only one 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 Runtime Library (RTL) call, and tracks the value of each of its arguments, as explained above. Once we obtain this information, we use the algorithm in Fig. 1 to interpret the data mapping semantics of each clause. The data mapping semantics can be classified into following categories,

- Copy In: A memory copy is introduced from the host to the corresponding list item in the device environment.
- Copy Out: A memory copy is introduced from the device to the host environment.

– Persistent Out: A device memory variable is not deleted, it is persistent on the device, and available to the subsequent device data environment.
– Persistent In: The memory variable is available on entry to the device data environment, from the last device invocation.

The examples in Subsect. 6.2 illustrate the above classification.

5.2 Baseline Memory Use Def Analysis

LLVM has an analysis called the MemorySSA [10], it is a relatively cheap analysis that provides an SSA based form for memory def-use and use-def chains. LLVM MemorySSA is a virtual IR, which maps `Instructions` to `MemoryAccess`, which is one of three kinds, `MemoryPhi`, `MemoryUse` and `MemoryDef`.

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. Figure 3 shows the LLVM Memory SSA for the OpenMP program in Listing 5.3. The comments in the listing denote the LLVM IR and also the corresponding `MemoryAccess`.

Listing 5.3. OpenMP program, for Fig. 3

```
1  int main(){
2    int A[10], B[10];
3    // 2 = MemoryPhi(1,3)
4    for (int i =0 ; i < 10 ; i++) {
5      // %arrayidx = getelementptr %A, 0, %
           idxprom
6      // store %i.0, %arrayidx,
7      // 3 = MemoryDef(2)
8      A[i] = i;
9    }
10 #pragma omp target enter data map(to:A
       [0:5])
11           map(alloc:B[0:10])
12 #pragma omp target
13   // 4 = MemoryPhi(2,5)
14   for (int i = 0 ; i < 10; i++) {
15     // %arrayidx7 = getelementptr %A, 0, %
           idxprom6
16     // %2 = load %arrayidx7
17     // MemoryUse(4)
18     int t = A[i];
19     // %arrayidx9 = getelementptr %B, 0, %
           idxprom8
20     // store %2, %arrayidx9
21     // 5 = MemoryDef(4)
22     B[i] = t
23   }
24
25   for (int i = 0 ; i < 10; i++) {
26     //arrayidx19 = getelementptr %B, 0, %
           idxprom18
27     //%3 = load %arrayidx19
28     // MemoryUse(4)
29     printf("%d",B[i]);
30   }
31
32     return 0;
33 }
```

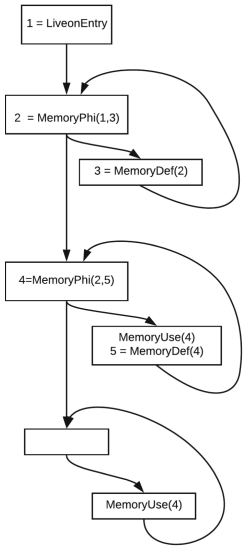


Fig. 3. Memory SSA of Listing 5.3

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 first node in Fig. 3 is a `LiveonEntry` node. The $3 = MemoryDef(2)$ node, denotes that there is a store instruction which clobbers the heap version 2, and generates heap 3, which represents the line 8 of the source code. Whenever more than one heap versions can reach a basic block, we need a $MemoryPhi$ node, for example, $2 = MemoryPhi(1, 3)$ corresponds to the for loop on line 4. There are two versions of the heap reaching this node, the heap 1, $1 = LiveonEntry$ and the other one from the back edge, heap 3, $3 = MemoryDef(2)$. The next `MemoryAccess`, $4 = MemoryPhi(2, 5)$, corresponds to the for loop at line 14. Again the clobbering accesses that reach it are 2 from the previous for loop and 5, from its loop body. The load of memory A on line 18, corresponds to the $MemoryUse(4)$, that notes that the last instruction that could clobber this read is `MemoryAccess` $4 = MemoryPhi(2, 5)$. Then, $5 = MemoryDef(4)$ clobbers the heap, to generate heap version 5. This corresponds to the write to array B on line 22. 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 29, corresponds to $MemoryUse(4)$, with the heap version 4, reaching this load. Since this loop does not update memory, there is no need for a `MemoryPhi` node for this loop, but we have left the node empty in the graph to denote the loop entry basic block.

Now, we can see the difference between the LLVM memory SSA (Fig. 3) and the array def-use chains required for our analysis (Fig. 2). We developed a dataflow analysis to extract the array def-use chains from the LLVM Memory SSA, by disambiguating the array variable that each load/store instruction refers to. So, for any store instruction, for example line 22, Listing 5.3, we can analyze the LLVM IR, and trace the value that the store instruction refers to, which is "B" as per the IR, comment of line 19.

We perform an analysis on the LLVM IR, which tracks the set of memory variables that each LLVM load/store instruction refers to. It is a context-sensitive and flow-sensitive iterative data flow analysis that associates each MemoryDef/MemoryUse with a set of memory variables. The result of this analysis is an array SSA form, for each array variable, to track its def-use chain, similar to the example in Fig. 2.

6 Evaluation and Case Studies

Listing 6.1. 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 6.2. 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];
```

For evaluating OMPSan we use the DRACC [1] suite, which is a benchmark for data race detection on accelerators, and also includes several data mapping errors also. Table 1 shows some distinct errors found by our tool in the benchmark [1] and the examples of Sect. 2. We were able to find the 15 known data mapping errors in the DRACC benchmark.

Listing 6.3. 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 6.4. 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){
```

Table 1. Errors found in the DRACC Benchmark and other examples

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

6.1 Analysis Time

To get an idea of the runtime overhead of our tool, we also measured the runtime of the analysis. Table 2 shows the time to run OMPSan, on few SPEC ACCEL and NAS parallel benchmarks. Due to the context and flow sensitive data flow analysis implemented in OMPSan, its analysis time can be significant; however the analysis time is less than or equal to the *-O3* compilation time in all cases.

6.2 Diagnostic Information

Another major use case for OMPSan, is to help OpenMP developers understand the data mapping behavior of their source code. For example, Listing 6.5 shows a code fragment from the benchmark "FT" in the "NAS" suite. Our tool can generate the following information diagnostic information on the current version of the data mapping clause.

Table 2. Time to run OMPSan

Benchmark name	-O3 compilation time (s)	OMPSan Runtime (s)
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

- *__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*

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

```
307 static void evolve(int d1, int d2, int d3)
308 {
309  int i, j, k;
311 #pragma omp target map (alloc: u0_real,u0_imag,u1_real,u1_imag,twiddle)
312  {
313 #pragma omp teams distribute
314   for (k = 0; k < d3; k++) {
315 #pragma omp parallel for
316    for (j = 0; j < d2; j++) {
317 #pragma omp simd
318     for (i = 0; i < d1; i++) {
319      u0_real[ ... ] = u0_real[ ... ]*twiddle[ ... ];
321      u0_imag[ ... ] = u0_imag[ ... ]*twiddle[ ... ];
```

6.3 Limitations

Since OMPSan is a static analysis tool, it includes a few limitations.

- Supports statically and dynamically allocated array variables, but cannot handle dynamic data structures like linked lists It can possibly be addressed in future through advanced static analysis techniques (like shape analysis).
- Cannot handle target regions inside recursive functions. It can possibly be addressed in future work by improving our context sensitive analysis.
- Can only handle compile time constant array sections, and constant loop bounds. We can handle runtime expressions, by adding static analysis support to compare the equivalence of two symbolic expressions.
- Cannot handle declare target since it requires analysis across LLVM modules.
- May report false positives for irregular array accesses, like if a small section of the array is updated, our analysis may assume that the entire array was updated. More expensive analysis like symbolic analysis can be used to improve the precision of the static analysis.
- 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 relies on some dynamic reference count mechanism. Runtime debugging approach will be required to handle such cases.

It is interesting to note that, we did not find any false positives for the benchmarks we evaluated on.

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. CGCM [6] was one of the first systems with static analysis to manage CPU-GPU communications. It was followed by [5], a dynamic tool for automatic CPU-GPU data management. The OpenMPC compiler [9] also proposed a static analysis to insert data transfers automatically. [8] proposed a directive based approach for specifying CPU-GPU memory transfers, which included compile-time/runtime methods to verify the correctness of the directives and also identified opportunities for performance optimization. [13] proposed a compiler analysis to detect potential stale accesses and uses a runtime to initiate transfers as necessary, for the X10 compiler. [11] has also worked on automatically inferring the OpenMP mapping clauses using some static analysis. OpenMP has also defined standards, OMPT and OMPD [3,4] which are APIs for performance and debugging tools. Archer [2] is another important work that combines static and dynamic techniques to identify data races in large OpenMP applications.

In this paper, we have developed OMPSan, a static analysis tool to interpret the semantics of the OpenMP map clause, and deduce the data transfers

introduced by the clause. Our algorithm tracks the reference count for individual variables to infer the effect of the data mapping clause on the host and device data environment. We have developed a data flow analysis, on top of LLVM memory SSA to capture the def-use information of Array variables. We use LLVM Scalar Evolution, to improve the precision of our analysis by estimating the range of locations accessed by a memory access. This enables the OMPSan to handle array sections also. Then OMPSan computes how the data mapping clauses modify the def-use chains of the baseline program, and use this information to validate if the data mapping in the OpenMP program respects the original def-use chains of the baseline sequential program. Finally OMPSan reports diagnostics, to help the developer debug and understand the usage of `map` clauses of their program. We believe the analysis presented in this paper is very powerful and can be developed further for data mapping optimizations also. We also plan to combine our static analysis with a dynamic debugging tool, that would enhance the performance of the dynamic tool and also address the limitations of the static analysis.

References

1. Aachen University: OpenMP Benchmark. https://github.com/RWTH-HPC/DRACC
2. Atzeni, S., et al.: Archer: effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 53–62, May 2016. https://doi.org/10.1109/IPDPS.2016.68
3. Eichenberger, A., et al.: OMPT and OMPD: OpenMP tools application programming interfaces for performance analysis and debugging. In: International Workshop on OpenMP (IWOMP 2013) (2013)
4. Eichenberger, A.E., et al.: OMPT: an OpenMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13
5. Jablin, T.B., Jablin, J.A., Prabhu, P., Liu, F., August, D.I.: Dynamically managed data for CPU-GPU architectures. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO 2012, pp. 165–174. ACM, New York (2012). https://doi.org/10.1145/2259016.2259038
6. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. SIGPLAN Not. **46**(6), 142–151 (2011). https://doi.org/10.1145/1993316.1993516
7. 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, POPL 1998, pp. 107–120. ACM, New York (1998). https://doi.org/10.1145/268946.268956
8. 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

9. 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, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010). https://doi.org/10.1109/SC.2010.36
10. LLVM: LLVM MemorySSA. https://llvm.org/docs/MemorySSA.html
11. Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., Pereira, F.M.Q.: DawnCC: automatic annotation for data parallelism and offloading. ACM Trans. Arch. Code Optim. **14**(2), 13:1–13:25 (2017). https://doi.org/10.1145/3084540
12. Novillo, D.: Memory SSA - a unified approach for sparsely representing memory operations. In: Proceedings of the GCC Developers' Summit (2007)
13. 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, PACT 2012, pp. 33–42. ACM, New York (2012). https://doi.org/10.1145/2370816.2370824