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
[2] IBM Toronto Laboratory

Abstract. OpenMP has made it convenient to port an existing application from CPU to heterogeneous systems like GPUs. Starting from 4.5 standard, target offloading features can be used to offload individual kernels to the GPU and other devices. Several users have noted that one of the most challenging and error-prone tasks is the memory management for the device offloading.
In this paper, we present a static analysis tool, OmpSan, a code sanitizer, that helps the developers understand the correct usage and performance implications of the target `map` clause. We present several case-studies to show the effectiveness of our powerful analysis. OmpSan can detect bugs resulting from incorrect usage of `map` clause, and also report diagnostic information and simple fixes for the bug.

Keywords: OpenMP Offloading · OpenMP Target Data Mapping · LLVM · Memory Management · Static Analysis · Verification · Debugging

1 Introduction

OpenMP is a widely used directive-based parallel programming model, that now 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 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 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.

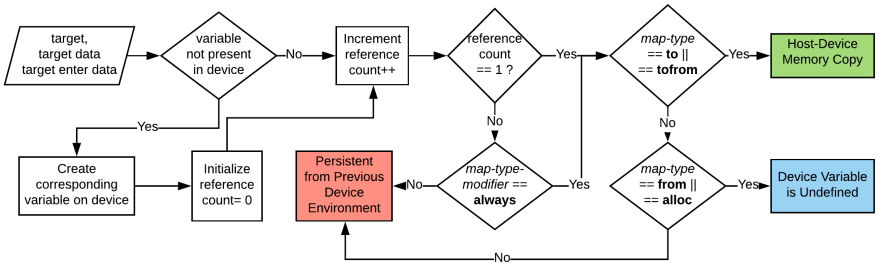
1.1 OpenMP 4.5 Map Semantics

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 standard. For correctness, in this paper we assume the device is a GPU, and mapping a variable from host to device introduces a host-device memory copy, and vice-versa. However, the bugs that we identify reflect errors in the OpenMP code regardless of the target device.

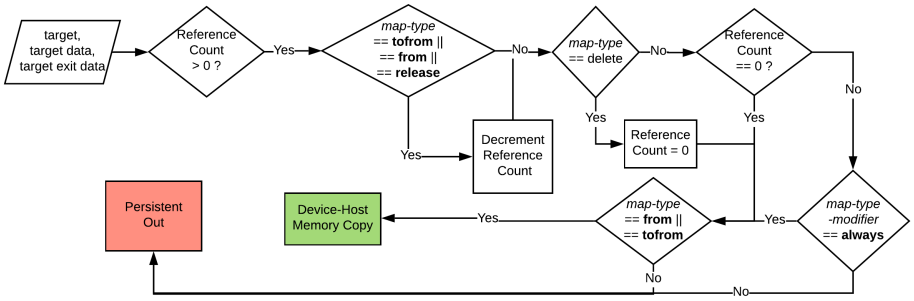
The different map types that OpenMP 4.5 supports are,

- 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

The default map type for arrays is *tofrom*, while the default for scalars is *first-private*, that is the only copy the value of the scalar at the entry to the device environment. As Figure 1 shows, OpenMP 4.5 specification uses the reference



(a) Flowchart for Enter Device Environment



(b) Flowchart for Exit Device Environment

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

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 Our Solution

To address the complexity of using OpenMP's target map clauses, we propose a static analysis tool called OmpSan to perform OpenMP code "sanitization". OmpSan is a compile-time tool, which does static verification of 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 dataflow information (reaching definitions via LLVM's memory SSA representation [7]) between the OpenMP and baseline code. We developed an LLVM-based implementation of our approach and evaluated its effectiveness using several case studies. Our major contributions include the fo1llowing.

- 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 static analysis technique to validate if the host/device memory copies respect the original memory def-use relations.
- 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 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 section 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 3 potential errors in user code arising from improper usage of the data mapping constructs.

2.1 Default Scalar Mapping

In Our first example, Listing 2.1, 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) exiting a data environment where the variable was mapped as "alloc" does not decrement the reference count, and a variable is mapped back from device to host only if the reference count is decremented to 0. We can track the reference count for "B" is as follows,

- Line 5, reference count = 1
- Line 6, enter data environment, reference count = 2
- Line 8, exit data environment "alloc", reference count = 2
- Line 9, exit data environment "from", reference count =1
- Line 12 accesses stale data of "B", since it was not mapped back to host

As Listing 2.4 shows, replacing "alloc" with "from" on line 6, will update the host version of "B" on exit of the map region at line 9.

(Note: This is no longer a bug in OpenMP 5.0, since even "alloc" decrements the reference counter)

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 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 report diagnostics and errors with possible fixes to help the developers in using the data mapping clauses.

Example 2 Listing 2.5 shows another example of a reference count issue. 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 of 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

```
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 Background

3.1 Memory SSA

Our analysis is based on the LLVM Memory SSA [7] [9], which is an imprecise implementation of Array SSA[4]. We construct the def-use chains for each array variable, based on the Memory SSA. LLVM Memory SSA is a virtual IR, where every definition and phi node creates a new version of memory, which are numbered. The Memory SSA IR has the following kinds of instructions/nodes,

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

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.
- A $MemoryDef$ node, clobbers the entire array associated with its store instruction.
- $MemoryPhi$ nodes 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

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, as chain of recurences. Then it can be used to symbolically evaluate the minimum and maximum value of every array index expression. section 5 has details of how we implement the analysis and handle different cases.

4 Our Approach

OmpSan assumes certain practical use cases, 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 with the algorithm and exemplify it with a concrete example to illustrate the algorithm in action.

4.1 Algorithm

The Algorithm 1 shows an overview of the data map analysis. Firstly, we collect all the array variables used in the various 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. Then, we call the function, `InterpretTargetClauses`, which modifies the Array SSA graph, as per 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 1 Let us consider the first example Figure 2a to illustrate our approach for analysis of data mapping clauses. `ConstructArraySSA` of Algorithm 1, constructs the memory SSA for arrays "A" and "C" as shown in Figure 2b. Then, `InterpretTargetClauses`, removes the edges between host and device nodes, as shown in Figure 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 Figure 2d. For example $L1$ is connected to $S2$ with a host-device memory copy for the enter data map pragma with `to`: $A[0:50]$ on line 5. Also, we connect $INIT$ node with $L2$, to account for the `alloc`:$C[0:100]$, which means an uninitialized reaching definition.

Lastly, `ValidateDataMap` function, traverses this graph, and we can make the following observations,

- Error, Node $S4$:$MemUse(5)$ is not reachable from its corresponding definition $L2:5 = MemPhi(0,6)$
- Warning, Only Partial array section $A[0:50]$, reachable from definition $L1:1 = MemPhi(0,2)$ to $S2:MemUse(1) < 0:100 >$

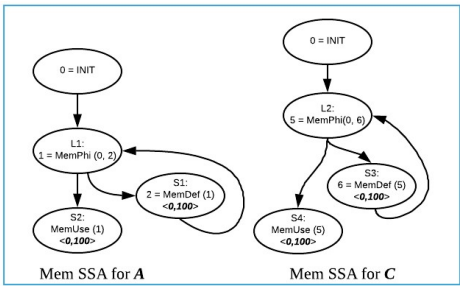
section 6 has several examples of the errors and warnings we report.

Algorithm 1 Overview of Data Mapping Analysis

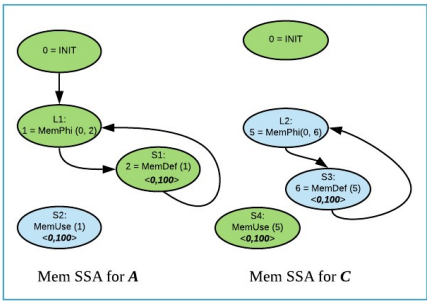
```
1: function DataMapAnalysis(Module)
2:     MappedArrayVars = φ
3:     for ArrayVar ∈ MapClauses do
4:         MappedArrayVars = MappedArrayVars ∪ ArrayVar
5:     ConstructArraySSA(Module, MappedArrayVars)
6:     InterpretTargetClauses(Module, MappedArrayVars)
7:     ValidateDataMap(MappedArrayVars)
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
```

```
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     }
```

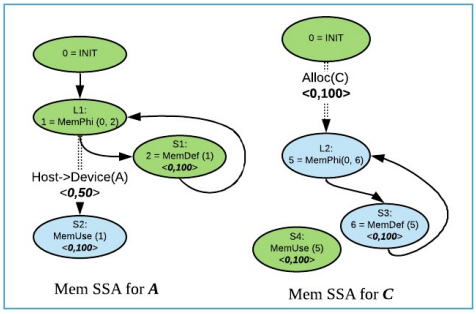
(a) Example 1, user Code



(b) Memory SSA



(c) Classify Host/Device Regions



(d) Host/Device Memory Copies

Fig. 2: Data Map Analysis Example 1

5 Implementation

We 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 def-use 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 def-use chains, explained

in 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

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 = {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 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 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,

- 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.

subsection 6.2 shows some examples, to illustrate the above classification.

5.2 Baseline Memory Use Def Analysis

Listing 5.3 shows an example, with Figure 3 as the simplified MemorySSA.

Listing 5.3: 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 }
```

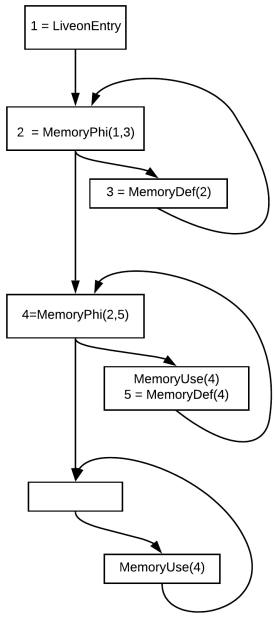


Fig. 3: MemorySSA

Now, given this MemorySSA, we can extract the Array def-use chains, if we can disambiguate the memory variable that the load/store instruction refers to. So, for any store instruction, for example line 18, we can analyze the LLVM IR, and trace the value that the store instruction refers to, which is "B" as per line 15.

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, for each array variable, to tracks its def-use chain.

Scalar Evolution Analysis, Array Sections We have 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 memory alloc instructions.

6 Evaluation and Case Studies

We use the the DRACC benchmark [1] to evaluate our tool. Table 1 shows some distinct errors found by our tool in the benchmark[1] and the examples of section 2. We were able to find all the data mapping errors in the DRACC benchmark.

Table 1: Errors found in the DRACC Benchmark

File Name	Error/Warning
DRACC File 22	ERROR Definition of :b on Line:18 is not reachable to Line:34, Missing Clause:to:Line:32
DRACC File 26	ERROR Definition of :c on Line:35 is not reachable to Line:46 Missing Clause:from/update:Line:44
DRACC File 30	ERROR Definition of :c on Line:25 is not reachable to Line:38 Missing Clause:to:Line:36
DRACC File 23	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

6.1 Time to Compile

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 runtime is significant, and almost as much as the time to compile the entire program with "-O3".

Table 2: 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

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 6.1 shows a code fragment from the benchmark "FT" of "NAS" suite. 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*

Listing 6.1: *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[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];
321           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];
```

6.3 Limitations

Since OMPSan is a static analysis tool, there are a few limitations as listed below,

- We can only deal with Array variables, and we cannot handle dynamic data structures like linked lists. This is an inherent limitation of static analysis
- Our implementation cannot handle target regions inside recursive functions, which can be fixed by improving our context sensitive analysis
- Our implementation can only handle compile time constant array sections, and constant loop bounds. Our tool can be extended to handle runtime expressions, if we can compare the equivalence of two symbolic expressions.
- Cannot handle "declare target", it requires analysis across LLVM modules.

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 [3] was one of the first systems with static analysis to manage CPU-GPU communications. It was followed by [2], a dynamic tool for automatic CPU-GPU data management. The OpenMPC compiler [6] 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 [5]. Pai et. al [10] proposed a compiler analysis to detect potential stale accesses and uses a runtime to initiate transfers as necessary, for the X10 compiler. [11] also proposed a static analysis technique to optimize the data transfers for the X10. [8] has also worked on automatically inferring the OpenMP mapping clauses using some static analysis.

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. It validates, if the data mapping respects the original def-use chains of the baseline program. Finally OMPSan reports diagnostics, to help the developer debug and understand the usage of `map` clauses of their program.

References

1. AachenUniversity: Openmp benchmark https://github.com/RWTH-HPC/DRACC
2. 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. pp. 165–174. CGO '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2259016.2259038, http://doi.acm.org/10.1145/2259016.2259038
3. 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 (Jun 2011). https://doi.org/10.1145/1993316.1993516, http://doi.acm.org/10.1145/1993316.1993516
4. 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
5. 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
6. 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
7. LLVM: Llvm memoryssa https://llvm.org/docs/MemorySSA.html
8. 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
9. Novillo, D.: Memory ssa- a unified approach for sparsely representing memory operations. In: Proceedings of the GCC Developers' Summit (2007)
10. 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
11. 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