Static Generation of Efficient OpenMP Offload Data Mappings

Luke Marzen
Department of Computer Science
Iowa State University
Ames, IA, USA
ljmarzen@iastate.edu

Akash Dutta
Department of Computer Science
Iowa State University
Ames, IA, USA
adutta@iastate.edu

Ali Jannesari
Department of Computer Science
Iowa State University
Ames, IA, USA
jannesar@iastate.edu

Abstract—Increasing heterogeneity in HPC architectures and compiler advancements have led to OpenMP being frequently used to enable computations on heterogeneous devices. However, the efficient movement of data on heterogeneous computing platforms is crucial for achieving high utilization. Programmers must explicitly map data between the host and connected accelerator devices to achieve efficient data movement. Ensuring efficient data transfer requires programmers to reason about complex data flow. This can be a laborious and error-prone process since the programmer must keep a mental model of data validity and lifetime spanning multiple data environments. We present a static analysis tool, `OMPDart` (**OpenMP** **Da**ta **R**eduction **T**ool), for OpenMP programs that models data dependencies between host and device regions and applies source code transformations to achieve efficient data transfer. Our evaluations on *nine* HPC benchmarks demonstrate that `OMPDart` is capable of generating effective data mapping constructs that substantially reduce data transfer between host and device.

Index Terms—High performance computing, Parallel programming, Data transfer, Static analysis, GPU, Performance optimization, Heterogeneous computing, OpenMP

I. INTRODUCTION

High-performance heterogeneous computing architectures have increasingly deployed accelerators such as GPUs. In recent years, the Top500 List [1] has been dominated by supercomputers leveraging GPU accelerators to achieve higher performance and efficiency. This paradigm shift offers unprecedented computational power, but it also brings forth challenges in efficiently managing data movement across diverse computing resources, as each connected device typically has its own distinct memory space. In this context, the efficient movement of data is paramount to achieving high performance, as redundant data transfers introduce overheads in terms of execution time and energy.

OpenMP is a portable parallel programming model that has been primarily used for easily parallelizing serial code. Over the past few years, it has been increasingly used to offload computations to accelerators. The implicit data-mapping rules that apply by default in OpenMP offload applications often lead to redundant data transfers. To address these challenges, programmers are required to explicitly define data mappings between the host (CPU) and connected accelerator devices (typically GPUs) using OpenMP's `target data` and `target update` constructs. Given that inefficient data transfer can dominate execution time, reducing data transfer while maintaining correctness stands out as a paramount optimization goal for developers. However, the complexity of the OpenMP specifications in combination with the necessity to reason about data flow across multiple memory spaces for each variable make this a nontrivial and time-consuming task.

In this work, we present static data-flow analysis techniques that include interprocedural and context-sensitive approaches to identify data dependencies between memory spaces. We introduce `OMPDart`, a static analysis tool for OpenMP programs designed to automate the data mapping process by modeling data dependencies between host and device regions and inserting data-mapping constructs in the source code to resolve them. Our techniques are based on a hybrid code representation that combines an Abstract Syntax Tree (AST) and a Control Flow Graph (CFG). Nine HPC benchmarks are used to evaluate the tool by comparing the performance of the tool-defined mappings against an expert hand-optimized implementation. We show that `OMPDart` can correctly generate data mappings that perform substantially better than default data mappings and on par with or, in some cases, better than the expert-defined mappings.

In this paper, we make the following contributions:

- Introduced static data-flow analysis techniques for identifying data dependencies between memory spaces.
- Implemented a hybrid AST-CFG representation to perform interprocedural and data-flow analysis to appropriately map data to memory spaces.
- Leveraged the Clang/LLVM toolchain to implement the proposed techniques in a source code transformation tool that enables the automatic generation of OpenMP data mapping directives.
- Experimentally evaluated our tool on nine HPC benchmarks and compared it against expert-defined data mappings, showing competitive performance with the expert implementations and a significant reduction in data transfer over the implicit OpenMP data mapping rules.
- Improved the performance of the OpenMP offload version of LULESH 2.0 in HeCBench by using `OMPDart` to identify alternative data mappings, resulting in an 85% reduction in data transfer and a 1.6× speedup over expert-defined mappings on a medium input size.

The remainder of this paper is organized as follows. Section II provides background on OpenMP offloading and static data-flow analysis. Section III provides a motivating example, followed by our proposed approach in Section IV and experimental setup and evaluations in Sections V and VI. We discuss our approach and results in Section VII and conclude the paper with Section IX.

II. BACKGROUND

A. OpenMP

`OpenMP` is a widely used shared-memory programming model for parallelizing serial code. In its simplest form, OpenMP requires users to simply insert `pragmas` before appropriate code blocks (such as loops) to distribute the workload between multiple processors. With the 4.0 specifications [2], OpenMP introduced a set of directives to enable compilers to offload work to devices such as GPUs. This helped abate dependence on vendor-specific compiler infrastructures and ease the use of GPUs for general-purpose computing. Offloading with OpenMP consists of two high level steps: (i) transfer and map data between host and device, (ii) offload computations to the device. Offloading computations to GPUs can potentially provide significant performance and energy efficiency improvements over traditional CPU-based multi-core processing. OpenMP offload directives enable effective exploitation of coarse-grained and scalable parallelism on GPUs. With the drastic improvement in GPU hardware, data mapping between host and device is now one of the primary bottlenecks in OpenMP offload computations. To this end, OpenMP introduced the `data map` clauses to allow programmers more fine-grained control over data transfer between host and device. However, this is not always straightforward to accomplish, especially without an in-depth understanding of an application or its runtime behavior. Our proposed tool aims to automate the selection of such clauses and help programmers improve performance by reducing data transfer overheads.

B. Static Data-Flow Analysis

Flow-sensitive analysis considers statement order in a program and tracks data constraints at each point of a program. For this, *Control Flow Graphs* (CFGs) are typically used. The nodes of a CFG consist of basic blocks and its directed edges represent the control flow paths between different basic blocks.

Path-sensitive analysis extends *flow-sensitive analysis* by considering all possible execution paths of a program. While *path-sensitive analysis* is more precise, it increases the time and space complexity since the CFG grows exponentially.

Context-sensitive analysis, sometimes referred to as *polyvariant analysis*, is a type of *interprocedural analysis* concerned with understanding the influence of calling contexts, i.e. the program state at function entry.

Liveness analysis, also known as *live-variable analysis*, is a type of data flow analysis that is used to determine which variables are *live* at a given point in the program's execution. A variable is considered *live* if it holds a value that may be used or referenced at some point in the program's

future execution. Common uses for *liveness analysis* include determining efficient register allocation during compilation and facilitating dead-code elimination.

III. MOTIVATION

This section will review common programming patterns that result in redundant data transfer. We will also demonstrate how programmers can easily introduce bugs while trying to minimize communication overhead, highlighting the error-prone nature of manually performing this endeavor.

Listing 1 depicts a kernel called repeatedly inside a loop. Since no explicit data mappings are present, the implicit data-mapping rules are applied. Relying on the implicit data-mapping rules results in data being copied to the device on entry of an OpenMP target region and copied from the device on exit. In Listing 1, this means coping array `a` to and from the device every iteration of the outer loop.

```
1   int a[N] = {};
2   for (int i = 0; i < N; ++i) {
3       #pragma omp target
4       for (int j = 0; j < N; ++j) {
5           a[j] += j;
6       }
7   }
```

Listing 1. Kernel nested inside a loop creates redundant data transfer each iteration of the outer loop.

```
1    int a[N] = {};
2    #pragma omp target
3    for (int i = 0; i < N; ++i) {
4        a[i] += i;
5    }
6
7    #pragma omp target
8    for (int i = 0; i < N; ++i) {
9        a[i] *= i;
10   }
```

Listing 2. Redundant data transfer between kernels due to reliance on implicit data-mapping rules.

Listing 2 depicts redundant data transfer between kernels. After execution of the first `for` loop, the default behavior is to copy array `a` back to the host then immediately back to the target, even though no data has been written to `a` in between the execution of these kernels. To reduce the communication overhead in Listings 1 and 2, programmers must explicitly map each variable between the host and accelerator devices, using OpenMP's `target data` and `target update` constructs. These simple examples represent patterns of redundant data transfer found in real world applications.

Listing 3 shows an incorrect data mapping. In this example the programmer intends for the array to be transferred from the device back to the host to be summed each iteration. However, the programmer noticed that the implicit data mappings would introduce data movement in both directions each iteration of the outer loop. In an attempt to eliminate the redundant data transfer, the programmer created a data region that moves array `a` before and after executing the outer loop, then created a second mapping on the first inner loop to transfer the data

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

Listing 3. Incorrect usage of the map construct.

back to the host for summation. The actual behavior of this code is to only transfer the data before and after executing the outer loop. This is because OpenMP 5.2 uses a reference count mechanism to decide when to copy data to and from a device map environment [3]. The reference count is incremented every time a new device map environment is created and decremented when exiting a region with the `from` or `release` map-type. Data is only actually copied from the device to the host when the reference count is decremented to zero. To get the expected behavior, the programmer should remove the inner loop data mapping and instead insert an `update from` clause directly after the first inner loop. It is clear from this initially benign example that the data mapping rules are more complicated than may initially meet the eye and necessitate a deep understanding to use efficiently and correctly. The complexity of this problem is exacerbated when considering the interactions of global variables and data mapping regions that map across multiple functions. This is obfuscated further when accessing the data mapping of many variables with less trivial control flow. Automating this task has the potential to improve the performance of applications on heterogeneous computing platforms while simultaneously reducing the time and effort required to create these programs. To the best of our knowledge, no prior works detail context- and flow-sensitive techniques that can correctly generate efficient OpenMP data mapping constructs for complex real-world programs.

IV. APPROACH

Our approach begins by leveraging the LLVM toolchain to parse the input source code and generate an AST. We then transform the AST into CFGs for each function and construct a hybrid data structure that integrates the AST of a function with the corresponding CFG, which we refer to as an *AST-CFG*. The AST is analyzed to identify operations that perform memory accesses. An interprocedural analysis is conducted on the AST-CFGs to augment each graph with the side effects of function calls. These side effects encompass any memory accesses to variables outside the local scope of a function, such as global variables and variables passed by reference. Next, we traverse the AST-CFG of each function to identify potential data dependencies between the host and device memory spaces and make determinations about how to most efficiently satisfy the dependencies using OpenMP data mapping directives.

TABLE I
AST NODES RECOGNIZED AS OFFLOAD KERNELS.

Clang AST Node	OpenMP Directive
OMPTargetDirective	`omp target`
OMPTargetParallelDirective	`omp target parallel`
OMPTargetParallelForDirective	`omp target parallel for`
OMPTargetParallelForSimdDirective	`omp target parallel for simd`
OMPTargetParallelGenericLoopDirective	`omp target parallel loop`
OMPTargetSimdDirective	`omp target simd`
OMPTargetTeamsDirective	`omp target teams`
OMPTargetTeamsDistributeDirective	`omp target teams distribute`
OMPTargetTeamsDistributeParallelForDirective	`omp target teams distribute parallel for`
OMPTargetTeamsDistributeParallelForSimdDirective	`omp target teams distribute parallel for simd`
OMPTargetTeamsDistributeSimdDirective	`omp target teams distribute`
OMPTargetTeamsGenericLoopDirective	`omp target teams loop`

Finally, the list of data mapping directives, along with the statements and variables they correspond to, are passed to the *Rewriter* stage (Section IV-F), which is responsible for consolidating and determining the final insertion location of each directive and outputting the transformed source code. An overview of our workflow is shown in Figure 1.

A. Input

The expected input is valid C/C++ source code with OpenMP offloading directives. This code should not include any instances of `target data` or `target update` directives. `OMPDart` aims to ease programmer effort and automatically insert the appropriate OpenMP data directives. It is important to acknowledge that in some cases a programmer may deliberately intend to use stale data. In such scenarios, our tool would be unfit as it lacks the capability to discern the programmer's intentions. However, the tool's output could serve as a helpful starting point that the programmer can manually modify to achieve their desired behavior.

B. Parsing the Abstract Syntax Tree

`OMPDart` takes a compilable OpenMP program as input and relies on its AST for further analyses. We use the LibTooling [4] library provided by Clang to build an AST for the input C/C++ code. Operating on the AST presents a more complex challenge in assessing memory accesses in comparison to LLVM *Intermediate Representation* (IR), but it helps maintain a closer relation to the source code. This relation makes analyzing ASTs more suitable for source-to-source transformations.

`OMPDart` begins by parsing the AST to identify memory accesses associated with each variable reference. The memory accesses are grouped by parent function and classified as read, write, read/write, or unknown. There are many cases in which
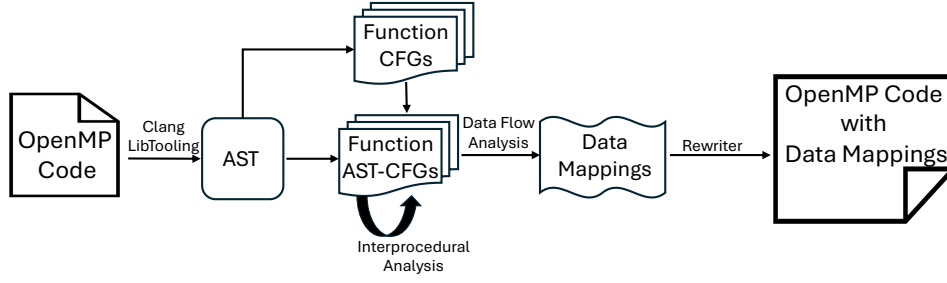
Fig. 1. Workflow used by `OMPDart` for identifying data dependencies and transforming source code for C and C++ OpenMP GPU offloading applications.


```
int foo(int a[]) {
    int x = bar(a);
    if (x > 0) {
        a[x] = 0;
    }
    return x;
}
```
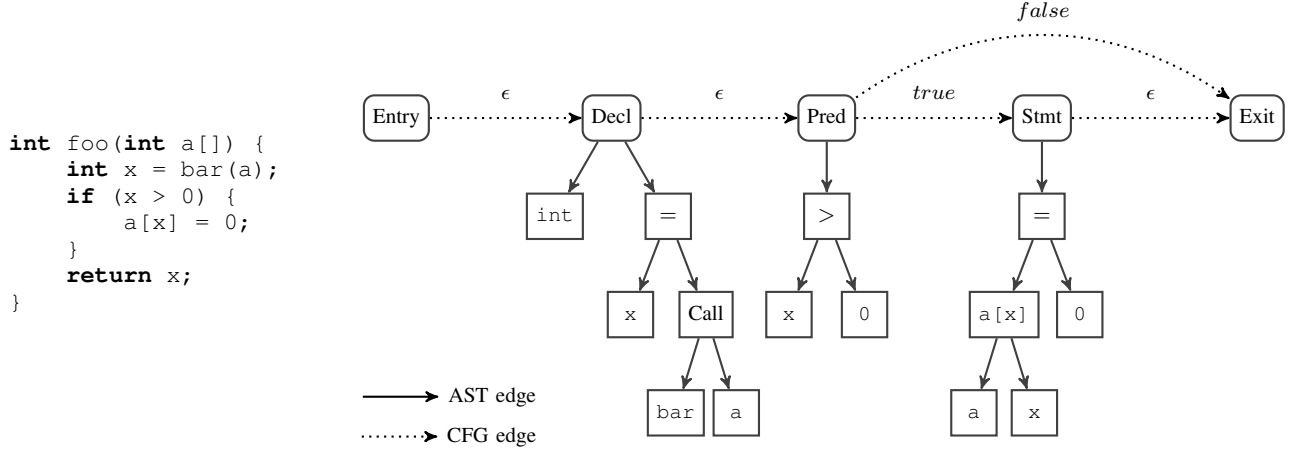
Fig. 2. Example of AST-CFG representation. The graphical figure on the right is the AST-CFG representation of the code on the left.

the type of memory access may be initially unknown, such as when a pointer is passed as a parameter to another function.

In certain scenarios, discerning memory accesses necessitates cross-translation unit analysis, particularly when a function definition resides in another translation unit. Currently, our tool is confined to analyzing individual translation units, so conservative assumptions are derived from declared function prototypes. We assume data passed via a pointer to `const` is strictly read-only, while non-static global variables and other parameters passed by pointer are handled conservatively.

In this stage, conditionals, loops, and target directives are identified and used to construct a CFG for each function definition in the input file. `IfStmt` and `SwitchStmt` AST nodes are classified as conditionals and `ForStmt`, `WhileStmt`, and `DoStmt` are AST nodes recognized as loops. Nodes of the CFG are marked as to whether they are offloaded or not, depending on whether they belong to an offloaded region. Offload kernels are blocks of code designated to be executed on a target device. A list of `OMPExecutableDirective` AST nodes recognized as offload kernels can be seen in Table I. This list includes all `target` directives except the `target (enter/exit) data` and `target update` directives.

The AST and CFG are combined to form a hybrid AST-CFG representation in which each node of the CFG is linked with the corresponding AST representation. To the best of our knowledge, this is a novel data structure, and its application in this context is unique. The AST-CFG is integral to our analysis since it maintains both the structural hierarchy of

the code represented by the AST and the control flow relationships of the CFG. A traversal of the CFG edges models data flow, variable liveness, and data validity to determine data dependencies, while intermittent AST analysis assists in detecting array access patterns that are crucial for deciding when to transfer data. Figure 2 shows example code and the corresponding AST-CFG representation. The AST-CFG is constructed on a function level, similar to the process of constructing a *Code Property Graph* (CPG). CPGs were first introduced by Yamaguchi et al. [5] for the purposes of detecting vulnerabilities in C code.

C. Interprocedural Analysis

Now that each procedure has been identified and has a CFG associated with it, an interprocedural analysis step is applied. This step analyzes how function calls affect the data of the caller. Specifically, we are interested in the passing of parameters and the use of common global variables across functions. The tool iteratively checks each function to determine if and in what manner any global variables or data at the address of pointer parameters were accessed. This includes whether data is read or written and if such accesses occur on the host or device. Once a determination is made about the use of pointer parameters or global variables, the model is augmented at each call site of the function with maximally pessimistic assumptions about the memory accesses of the callee. In this context, a maximally pessimistic approach means the analysis assumes the worst-case scenario for any unknown or uncertain

TABLE II
OpenMP Constructs That OMPDart Will Insert to Resolve Data
Dependencies

OpenMP Construct	Description
map(to:)	on region entry copies data from host to device
map(from:)	on region exit copies data from device to host
map(tofrom:)	on region entry copies data from host to device and on exit copies data from device to host
map(alloc:)	on region entry allocates memory on device
update to()	updates data on device with the value from host
update from()	updates data on host with the value from device
firstprivate()	on region entry initializes a private copy on the device with the original value from the host

information. This conservative approach helps to ensure that we don't violate the correctness of a program, of course, with the potential downside that we may miss an optimal solution. This is particularly relevant in cases when we don't have visibility into a function that may access memory in unknown ways. This process can be repeated several times up to the maximum call depth of any function. Each pass provides new information about the function to the model, but can be stopped early if no updates are made during a pass. This helps the model make more accurate assumptions about data accesses.

Function inlining is an alternative approach that could be considered here. The main advantage would be the potential to avoid interprocedurally invalid paths. However, applying inlining can cause CFG size to balloon exponentially. In addition, inlining cannot handle mutually recursive functions on its own. Heuristics could be used to decide when to apply inlining to manage this trade-off between precision and CFG size. Even without employing an inlining approach, effective decisions can still be made reliably, as shown in section VI.

D. Data-Flow Analysis

The analysis is performed as a forward traversal of the AST-CFG for each function. We trace the reads and writes to any variable referenced inside any offloaded region with the goal of identifying data dependencies between the host and device memory spaces. Anti (Write After Read, WAR) dependencies and output (Write After Write, WAW) dependencies do not necessitate any communication. Only true (Read After Write, RAW) dependencies need to be resolved by inserting one of the constructs described in Table II.

We define data to be *valid* in a memory space if the data was last written to in said memory space and *invalid* or *stale* if the data was last written to in any other memory space. While traversing the CFG, we track whether a memory space has a valid, up-to-date copy of each variable at each node. If a node is reached that writes to the host environment, the corresponding data on the target is marked as stale. If a node is reached that may read stale data, we have identified a true dependency and must evaluate the most efficient way to satisfy it.

For each function with at least one true dependency, we create a single target data region that encompasses all the kernels in the function's body. The starting point of the region is determined by finding the start of the earliest offload kernel, and the end location is the end of the last offload kernel in the function. However, doing this for a function consisting of a loop containing offload kernels would necessitate data being moved every iteration, possibly dominating execution time. To avoid this and maximize the opportunity for data reuse, we must extend the target data region to begin before any loop capturing the first kernel and end after any loop capturing the last kernel. Having a single target data region for each function enables general data mapping solutions since we can efficiently satisfy any dependencies between this data region and the rest of the function by using map(to/from/tofrom:) constructs and using a combination of map(alloc:) and update constructs to resolve dependencies between kernels and across loop iterations. A single data region introduces the additional requirement that any variable declaration in the function body used by both the host and device must precede the location at which the tool intends the placement of the target data region. If the input program violates this, the tool will detect this and issue an error indicating before which point the programmer should move the declaration so that the tool can continue successfully.

Identifying the ideal insertion point for OpenMP data mapping constructs is crucial for minimizing data transfer, as Section IV-E demonstrates. Misplacing a data construct in a loop when it could be placed outside the loop body will almost definitely incur a significant performance penalty. With the single target data region per function strategy that we have established, the difficulty of this problem becomes identifying correct and efficient placements of update constructs.

Upon encountring a branch in the CFG, each path is explored until they converge again or the function exits. Nodes are only explored once since if we reach a visited node, we know we have found a loop and need to make any data valid as it was prior to the already visited node. For instance, if data is valid on a device upon entering a loop, it should be valid at the end of the loop body so that each memory space is set up for the next iteration of the loop. If a node that is part of a loop needs updated data from another environment, we make note until we reach the end of the loop. If, by the end of the loop body, the data from the earlier node's environment is still valid, a data dependency does not exist between loop iterations, and we can safely map the data at a location prior to the loop. Otherwise, there is a data dependency between loop iterations and an update statement must be inserted somewhere in the loop body to preserve correctness.

For some special cases, we can employ the firstprivate clause to improve efficiency over the update to construct. The firstprivate clause creates private copies of variables for each thread and initializes them with the value from the master thread. When a firstprivate scalar variable is used within an OpenMP offload directive that launches a kernel, it may pass the initial

```
1   #define N 100
2   int main() {
3       int a[N];
4       ..
5       #pragma omp target teams distribute \
6               parallel for
7       for (int i = 0; i < N/2; i++) {
8           a[i] = i;
9       }
10      ..
11  }
```

Listing 4. Sample C code consisting of a `for` loop with OpenMP target construct used to generate the AST in Listing 5.

```
1   ForStmt
2   |-DeclStmt ..
3   | `-VarDecl .. used i 'int' cinit
4   |   `-IntegerLiteral .. 'int' 0
5   |-<<<NULL>>>
6   |-BinaryOperator .. 'bool' '<' ..
7   | |-DeclRefExpr .. 'i' 'int'
8   | `-BinaryOperator .. 'int' '/'
9   |   |-IntegerLiteral .. 'int' 100
10  |   `-IntegerLiteral .. 'int' 2
11  |-UnaryOperator .. 'int' postfix '++'
12  | `-DeclRefExpr .. 'i' 'int'
13  `-CompoundStmt ..
14    `-BinaryOperator .. 'int' lvalue '='
15      |-ArraySubscriptExpr ..
16      | |-DeclRefExpr .. 'a' ..
17      | `-DeclRefExpr .. 'i' ..
18      `-DeclRefExpr .. 'i' 'int'
```

Listing 5. Excerpt of the Clang AST dump generated from the source code in Listing 4 by running the command `clang -Xclang -ast-dump -fsyntax-only file.c`. Addresses, source locations, and other information has been omitted for readability, indicated by "..".

```
1   // partial_sum: valid on device, invalid on host
2   for (int j = 1; j <= hid; j++) {
3       sum = 0.0;
4       for (int k = 0; k < num_blocks; k++) {
5           sum += partial_sum[k * hid + j-1];
6       }
7       sum += input_weights[0][j];
8       hidden_units[j] = 1.0 / (1.0 + exp(-sum));
9   }
```

Listing 6. Code snippet from the HeCBench's OpenMP offloading implementation of the Rodinia backprop benchmark [10] with a nested `for` loop which is executed on the host and calculates a sum.

value from the host as an argument. This approach avoids additional memory allocations and copies when compared to using the `map(to:)` clause, as described in the Intel oneAPI GPU Optimization Guide [6]. Therefore, it is advisable to use the `firstprivate` clause for transferring read-only scalar variables rather than the map clause. It's important to note that this argument passing behavior is not explicitly defined in the OpenMP 5.2 Specification [3], but is rather left to the discretion of the compiler implementation and may further depend on the architecture of the target accelerator. While the referenced Intel Optimization Guide [6] specifically addresses the icx implementation, we have additionally verified that this behavior applies to both clang 17.0.4 and gcc 13.2.1 on Nvidia GPUs. This can be verified with the NVIDIA NSight System profilers [7], [8] to observe the reduction of host to device memcpys when using `firstprivate` over `map(to:)`. This is a specialized performance optimization that may not be widely known. By using this tool, programmers can take advantage of such lesser-known performance optimizations. When implementing this optimization, the analysis must be revised so that at the end of the kernel region, the model's internal state is updated to reflect that the `firstprivate` data is no longer valid.

Upon reaching the end of the target data region for a function, the problem becomes a liveness problem. For variables used in an offloaded region, we want to determine if they are subsequently read, since if read after the target region we must make sure that data will be valid upon region exit. If stale data may be read on the host after exit from a target data region, the `from` map-type is included in the variable's mapping clause to ensure the data will be valid.

E. Array Access Pattern Analysis

In [9], Guo et al. presented a compile-time algorithm that operates on the Clang AST and can successfully eliminate the transfer of unused array segments in C and C++ OpenMP programs. The algorithm makes determinations about what data transfer is truly necessary based on the analysis of `for` loops to determine which array bounds are accessed. This analysis may be impeded if the programmer's use of a `for` statement lacks any of the initializing, conditional, or incrementing/decrementing statements, or if any of these statements are overly complex. Furthermore, the algorithm presented in [9] has no mechanism to work with nested loops

and uses the lifetime of a for loop as the lifetime to map the data. In this section, we describe how we modify their approach for the purposes of arrays with multiple dimensions and nested loops, and how we determine efficient placement of update directives in scenarios involving nested loops.

Listing 5 shows the AST dump of the source code in Listing 4. In Listing 5, each component of the `for` loop is recognizable. Lines 2 through 4 are the initialization statement which represents `int i = 0` from line 7 of the source code. We know 0 is the lower bound; since the iteration statement is a unary increment on `i` (AST lines 11-12) and the conditional statement has `i` on the left-hand side of the less than operator (AST lines 6-7). The upper bound can be found by evaluating the expression on the right-hand side of the less than operator, `100/2` (AST lines 6,8-10), and subtracting 1 to avoid an off-by-one error.

`OMPDart` extends this bounds analysis to infer the access patterns in nested loops to assist the placement of OpenMP `update` directives. Consider the nested `for` loop from HeCBench's OpenMP offloading implementation of the Rodinia backprop benchmark [11] shown in Listing 6. The code is executed on the host and reads from the array `partial_sum`, which happens to be invalid on the host upon entering the snippet. Using the logic presented in Section IV-D, the automated tool will correctly identify that somewhere preceding line 5, it must insert the following OpenMP directive: `target update from(partial_sum)`. The

immediate choice would be to insert the statement between lines 4 and 5, however, this would cause the data transfer to occur every iteration of the inner loop. To avoid this redundant data transfer, the `update` directive should be placed before the outermost loop (preceding line 2). While both the options would produce technically correct code, in this example, placing the `update` directive in the inner loop would result in over 2GB of data transfer compared to less than 5MB when the directive is inserted before the `for` loops (with 65536 input elements). By correctly identifying the nested loop structure and access pattern of `partial_sum`, OMPDart can make a better decision on the placement of the `update` directive, which translates to a $14\times$ speedup.

Algorithm 1 Determining placement of `target update to/from()` OpenMP directives for array accesses in nested loops of arbitrary depth.

> **procedure** FINDUPDATEINSERTLOC($a, loops, locLim$)
> $idxExpr \leftarrow a.arraySubscript.idx$
> $indexingVars \leftarrow getReferencedVars(idxExpr)$
> $pos \leftarrow a$
>   **while** $loops$ is not empty **do**
> $forStmt \leftarrow loops.pop()$
>   **if** $forStmt$ is before $locLim$ in file **then**
>   **break**
>   **end if**
> $forIdxVar \leftarrow findIndexingVar(forStmt)$
>   **if** $forIdxVar$ is not a valid variable **then**
>   **continue**
>   **end if**
>   **if** $indexingVars.contains(forIdxVar)$ **then**
> $pos \leftarrow forStmt$
>   **end if**
>   **end while**
>   **return** pos
> **end procedure**

Algorithm 1 can be used to find the outermost loop that impacts the indexing of a given array. $loops$ is a stack that contains references to `for` statements. The top of the stack corresponds to the `for` loop that includes array access a in its body, with each subsequent element representing the enclosing loop. $locLim$ stores a statement which the directive must not precede, typically being the end of the preceding target kernel's scope. The procedure returns a pointer to the statement that the directive should directly precede or follow depending on whether the direction of data movement is `from` or `to` the device, respectively. From there, this location is used to further decide where the best placement of the `update to/from` statement might be.

F. Rewriter

The primary task of the rewriter is to insert the data mapping directives/clauses determined by the model in the previous stage. Table II shows the OpenMP constructs that the rewriter is responsible for inserting. For this the rewriter must realize the exact insertion point in the source code corresponding to the indicated statement AST node from the model.

When only a single offload region exists in a function, and the beginning of the offload region is the insertion point for `target data` directive, the rewriter can simply append a `map` clause to the existing target directive. Otherwise, the rewriter will insert a new `target data` directive and increase the indentation of the captured block. `firstprivate` clauses are simply appended to an existing OpenMP directive indicated by the model. Determining the exact insertion point in the source code for `target update` directives requires additional work. The previous stage provides a pointer to a `Stmt` node, which must be preceded/followed by a `target update` directive, however, this may or may not be a semi-terminated statement. The logic varies slightly when determining the insertion point for an `update to` directive compared to an `update from` directive. Generally, `update from` directives should be inserted before the statement indicated by the model from the previous stage and after for `update to` directives. Special considerations are made for statements contained in a loop conditional. `update to` directives pertaining to a memory access in a loop conditional should be placed at the beginning of the loop body. `update from` directives for `for`/`while` conditionals may need to be placed either before the loop or immediately preceding the end of the loop body. Since `do` conditionals occur at the end of the loop, `update from` directives will always be inserted at the end of the loop body. Finally, prior to inserting the directives and clauses into the source code, each type of directive and clause is consolidated based on their insertion point. This process condenses the constructs into a directive per insertion point. Finally, the directives are inserted into the source code at the determined locations.

V. BENCHMARKS

To evaluate OMPDart, *nine* OpenMP GPU programs/benchmarks were studied. Four benchmarks were chosen from the Rodinia GPU suite [21], and four from the HeCBench suite [10], detailed in Table III. The selection aimed to cover a range of complexities, with a preference for benchmarks featuring a mix of multiple kernels, numerous mapped variables, several `map` clauses, and the inclusion of `update` clauses.

Table IV details the complexities of each of the selected benchmarks, including the number of kernel regions, lines of code in offloaded regions, number of mapped variables, and an approximation of the number of possible mappings. The number of possible mappings is approximated by the sum of two parts. (1) The total combinations of mapping clauses. Each variable in each kernel have one of the **4** map-type types in $\{$`to`, `from`, `tofrom`, `alloc`$\}$. (2) The total combinations of `update` clauses. `update` clauses can be placed between every statement and there are **3** options for each variable which are `to`, `from`, or not to insert an `update` directive at this location. We divide the number of lines by **2** since this provides a more conservative representation of the actual number of effective placements when taking into account that

Application Name	Benchmark Suite	Domain	Description
Accuracy of Prediction (accuracy)	HeCBench	Machine Learning	Computes the classification accuracy of a neural network [12]
Allen-Cahn Equation (ace)	HeCBench	Fluid Dynamics	Phase-field simulation of dendritic solidification [13]
Back Propagation (backprop)	Rodinia	Pattern Recognition	Machine learning algorithm that trains the weights of connecting nodes on a neural network [14]
Breadth First Search (bfs)	Rodinia	Graph Traversal	Traverses all the connected components in a graph [15]
Coulombic Potential (clenergy)	HeCBench	Physics Simulation	Evaluates electrostatic potentials on a 3-D lattice using direct Coulomb summation method [16]
HotSpot (hotspot)	Rodinia	Physics Simulation	Thermal simulation tool used for estimating processor temperature based on an architectural floor plan and simulated power measurements [17]
LULESH (lulesh)	HeCBench	Hydrodynamics	Proxy application that simulates shock hydrodynamics [18]
Needleman-Wunsch (nw)	Rodinia	Bioinformatics	Non-linear global optimization method for DNA sequence alignments [19]
XSBench (xsbench)	HeCBench	Neutron Transport	Mini-app representing a key computational kernel of the Monte-Carlo neutron transport algorithm [20]

TABLE IV
COMPARISON OF BENCHMARK DATA MAPPING COMPLEXITY

Benchmark	Kernels	Offloaded Lines	Mapped Variables	Possible Mappings
accuracy	1	37	5	297
ace	6	202	7	2289
backprop	2	87	7	969
bfs	2	40	8	544
clenergy	2	103	5	812
hotspot	1	85	15	1972
lulesh	15	1293	65	129967
nw	2	122	12	2292
xsbench	1	271	8	3284

some statements span multiple lines, some lines are empty, and some only contain comments. The number of possible mappings is calculated as follows:

$$\text{mappings} = \text{kernels} \times \text{variables} \times 4 + \left\lfloor \frac{\text{lines}}{2} \right\rfloor \times \text{variables} \times 3$$

To evaluate OMPDart, three versions of each of the selected benchmark applications were evaluated.

1) *Unoptimized* – This version is devoid of any explicit data mappings and falls back on the default compiler-defined handling of data transfer and reuse.
2) *OMPDart Optimized* – The unoptimized version of each program was fed to OMPDart, and the output is the tool's optimized version of the program.
3) *Expert Optimized* – This is a version of the applications as implemented in HPC benchmark suites Rodinia and HeCBench. All applications used for evaluation had explicit data mappings in them to help improve bottlenecks with data transfer and synchronization.

VI. EXPERIMENTS

This section highlights our experimental evaluations of OMPDart and compares its outputs with the unoptimized OpenMP offload code and the expert-implemented versions with explicit data mappings (Section V).

Setup. We first execute and evaluate metrics for the unoptimized and expert versions of the OpenMP target offload codes. Next, we pass the unoptimized application through OMPDart

to output data mappings that substantially reduce the data transfer overhead. Our results were obtained on an NVIDIA A100 running CUDA 11.8.89 compiled with Clang 17.0.4.

Results and Analysis.

The OpenMP applications with data mappings generated by OMPDart were compiled with the same options as used for the unoptimized and expert implementations. The compiled application was then executed to collect the runtimes. Each application was also profiled with NVIDIA Nsight Systems (*nsys*) [8] to evaluate the number of **H**ost-**to**-**D**evice (HtoD) and **D**evice-**to**-**H**ost (DtoH) CUDA memcpy calls, the bytes transferred each way, and the total time taken by data transfer.

The data mappings generated by OMPDart significantly reduce the data transferred for all applications over the unoptimized versions as can be seen in Figure 3. Figure 4 shows that OMPDart successfully reduced GPU data transfer activity in terms of CUDA memcpy calls below the level of the expert mappings in 6 of the benchmarks. Figure 5 shows the speedup of each benchmark application with respect to the unoptimized implementation. For each application, the mappings were always *at least* as good as the expert implementations.

For ace, accuracy, and backprop, OMPDart produced data mappings identical to the expert-optimized program. For clenergy, the tool-generated mappings were equivalent to the expert mappings except for one additional mapping for a struct that the expert presumably had overlooked. This translates to a drastic 66% reduction in CUDA memcpy calls, but since the size of the struct is relatively small, there is only a reduction of 60kB in total data transfer and a marginal $1.01\times$ speedup over the expert implementation. The mappings generated for bfs differed from those of the expert due to the tool opting to resolve data dependencies through separate update to and update from clauses, whereas the expert version employed a single map clause to achieve an equivalent outcome. The tool employed the use of the firstprivate clause to reduce the number of CUDA memcpy calls compared to the expert-optimized program in hotspot, nw, and xsbench. As can be observed in Figure 4, this amounts to a reduction in CUDA memcpy calls by 57% in hotspot, 33% in nw, and 38% in xsbench over the expert defined mappings.
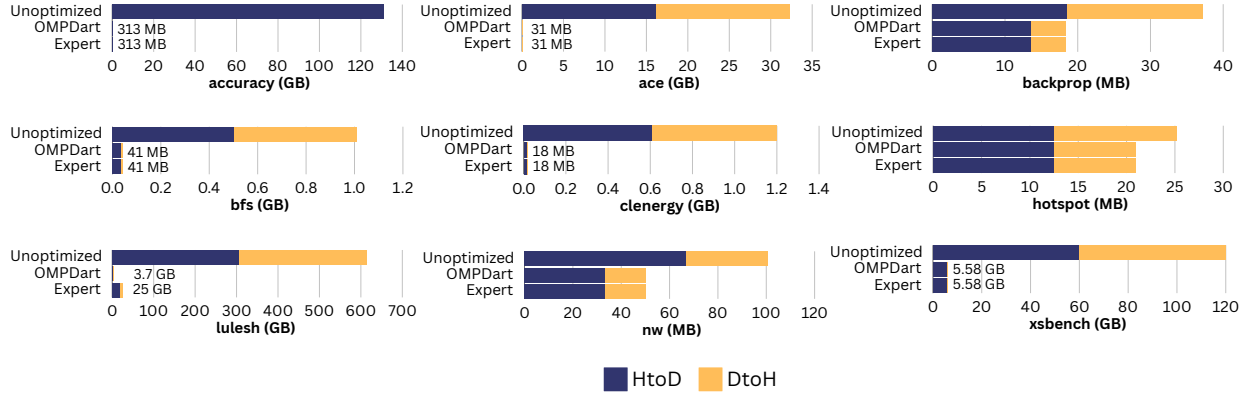
Fig. 3. Comparison of GPU data transfer activity (bytes) (lower is better). **HtoD**: Data transfer from CPU to GPU. **DtoH**: Data transfer from GPU to CPU.
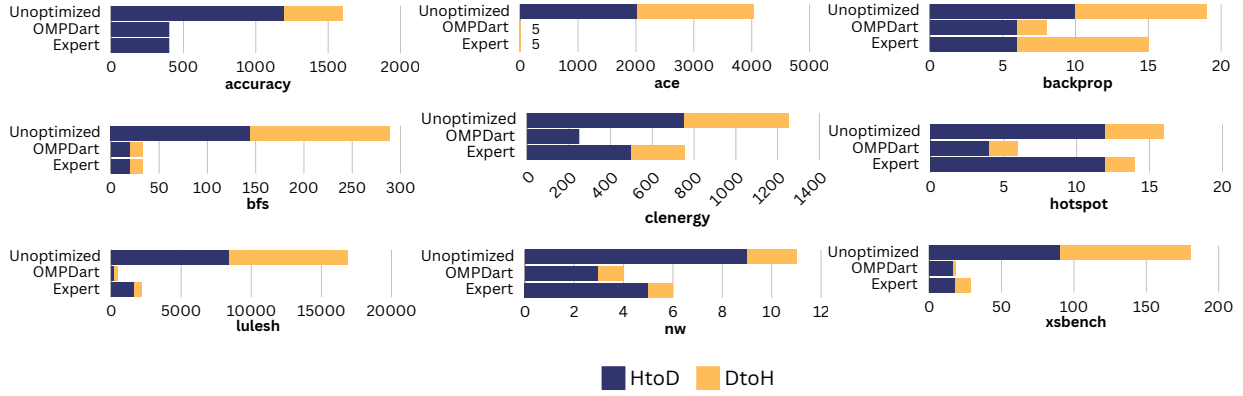

Fig. 4. Comparison of GPU data transfer activity (# calls) (lower is better). **HtoD**: Data transfer from CPU to GPU. **DtoH**: Data transfer from GPU to CPU.
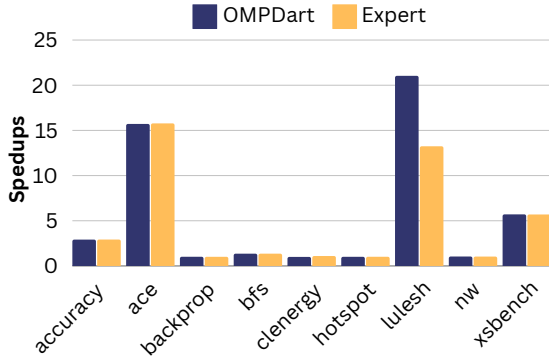

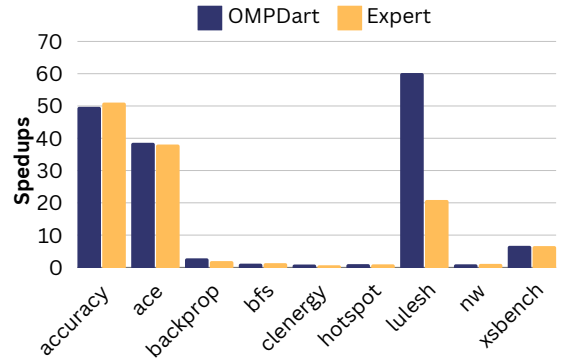Fig. 5. Speedups over unoptimized OpenMP offload code. (Higher is better)

Fig. 6. Improvements in data transfer wall times over unoptimized data mappings. (Higher is better)

Across these benchmarks, OMPDart reduces the data transferred between host and device by $1010\times$, $400\times$, $2\times$, $65\times$, $23\times$, $1.2\times$, $2\times$, and $20\times$ over default OpenMP offload data mappings for ace, accuracy, backprop, clenergy, bfs, hotspot, nw, and xsbench. OMPDart helps improve runtime performance by $16\times$, $2.9\times$, $1.01\times$, $1.11\times$, $1.36\times$, $1.01\times$, $1.04\times$, and $5.7\times$ for these applications respectively.

OMPDart generated mappings significantly outperformed the expert-defined mappings in lulesh, achieving a speedup of $1.6\times$ and a reduction in data transfer of over 23GB. This improvement can be primarily attributed to the inclusion of

several redundant update directives in the expert implementation. OMPDart helps remove such redundant mappings, which reduces the data transfer bottleneck and decreases synchronization overheads. Data mapping selections made by OMPDart help reduce host-to-device (HtoD) data transfers by $7.4\times$ over expert implementation and help reduce device-to-host (DtoH) data transfers by $5.1\times$. lulesh uses 65 variables across the host environment and 15 kernels over thousands of lines of code. Manually tracing their use and identifying the best implementation is an extremely time-consuming and challenging task. Such complex applications

have high potential for programmers to unintentionally create redundant data transfer, especially since compilers do not generate warnings for this. Our tool can go a long way in trying to help address such challenges.

As mentioned in Section IV-A, programmers might sometimes deliberately use stale data. The only benchmark studied that relies on the use of stale data is `backprop`. After running the tool on `backprop` an additional update directive was added by hand to adhere to this program behavior. All other mappings were generated automatically.

The tool's execution time is negligible. The average execution time of `OMPDart` on the benchmarks studied was 0.29s. `lulesh`, with 15 kernels, was the biggest benchmark evaluated and had the greatest overhead of 1.35s. Table V lists the tool's execution time to analyze and transform each benchmark.

Overall, across all studied benchmark applications, `OMPDart` generated mappings produce a geometric mean speedup of $2.8\times$ over the default implicit data mapping rules and $1.05\times$ speedup over the expert-defined mappings. A geometric mean reduction of 2.1GB in data transfer is saved by the tool over the default implicit data mapping rules. These results further affirm the critical importance of defining efficient data mappings. Furthermore, our evaluations show the impact that data transfer has on the overall application performance. To demonstrate this, we have also analyzed the wall times spent by each application on data transfer and synchronization. Almost all applications are better than the unoptimized implementation when `OMPDart` and expert mappings are evaluated on this metric. As seen in Figure 6, time taken for data transfer improves by up to $60\times$ and across all applications improves by a geometric mean of $5.1\times$ when `OMPDart` mappings are used. The improvements in time taken to transfer data and synchronize are similar in most cases for the expert implementation. Only in `lulesh` do the expert mappings perform significantly worse on this metric compared to `OMPDart`, which offers improvements in data transfer time of $20\times$. Overall, for all applications the geometric mean improvements for the expert implementation is $4.2\times$.

Each program with the mappings generated by `OMPDart` was also evaluated for correctness by comparing its output to the output produced by the expert implementation. The mappings generated by our tool produced outputs consistent with those produced by experts.

VII. DISCUSSION

`OMPDart` is a source-to-source transformation tool which makes these optimizations compiler agnostic. OpenMP offloading support and performance varies drastically between compilers and architectures [22]–[25]. Some compilers, oftentimes hardware vendor-specific compilers, only support a small set of architectures but incorporate many architecture-specific optimizations that may make a compiler the best option for offloading to certain targets. By implementing general data mapping optimizations on a source-code level we ensure portability and enable the programmer to use the compiler best suited to the target system.

Static analysis faces inherent limitations due to foundational results in computability theory, notably Rice's Theorem [26] and Turing's introduction of the Halting Problem [27]. The implications for static analysis necessitate conservative assumptions and approximations, which for static data-flow analyses means balancing precision and soundness. Our analysis errs on the side of soundness.

We assume that pointers can be disambiguated through alias analysis. If alias analysis fails to determine whether two pointers in a program can refer to the same memory location, the analysis will fail. Moreover, when there is a single element access of an array it is conservatively assumed to be the access of the entire array. This can be a source of correctness violation and can be mitigated by more precisely tracking accesses for array elements and using more sophisticated array access bounds analysis.

When using `OMPDart` it should be noted that it is currently limited to analyzing a single translation unit at a time. It performs best when it has full visibility into all functions. Functions defined in external translation units require conservative, worst-case assumptions about the behavior of those functions. In the future, the interprocedural analysis could be expanded to enable improved analysis capabilities for programs organized with kernels in separate translation units. The array bounds analysis presented in Section IV-E is currently only implemented for `for` loops, however, in the future this logic could be extended to encompass `while` and `do` loops. One approach for doing this would be to analyze the code preceding the loop to determine initial conditions, using the conditional statement of the `while`/`do` statement, and analyzing the loop body for possible incrementing statements.

At the time of our evaluation, AMD GPUs were not readily available to us, hence they were not tested. However, we believe a similar reduction in data transfers should be seen on AMD GPUs as well.

VIII. RELATED WORK

Mishra et al. [28], presented a tool that uses the Clang AST to identify nearby OpenMP offload kernels and create data mappings across multiple kernels. The program analysis presented here did not consider data dependencies between

kernel regions or loop iterations and cannot insert `update` or `firstprivate` constructs. Unlike `OMPDart`, this study only considered data mappings within the same function body.

Another work, OpenMP Advisor [29], utilizes machine learning to predict performant OpenMP offloading directives. As part of this work, the authors used a Data Transfer and Reuse Analysis Tool they had previously developed to predict the data mappings for programs before running it through the machine learning model that they developed.

Recently, Guo et al. [9] developed compile-time data transfer optimizations that center around filtering the transfer of unused array segments. They accomplish this automatically at compile time-based on analysis of the Clang AST. Guo et al. present techniques for array-bound analysis based on loop structures which we have used and extended in Section IV-E.

OMPSan (OpenMP Sanitizer) [30] is a static analysis tool that verifies the correctness of and identifies bugs related to OpenMP data mapping constructs. OmpMemOpt [31] employs a compile-time analysis that aims to identify and eliminate redundant data movements. This is accomplished by formulating a partial redundancy elimination problem and applying the lazy code motion compiler optimization technique. `OMPDart` differs from OmpMemOpt in that we focus on generating efficient data mappings while OmpMemOpt offers compiler optimizations to automate the removal of redundant memory copies at compile time. Due to the unavailability of OmpMemOpt, direct performance comparisons could not be made.

IX. CONCLUSION

In this work, we addressed the time-consuming and error-prone task of manually defining efficient OpenMP data mappings. The efficient movement of data is necessary for maximizing the utilization of modern heterogeneous computing platforms. We proposed a static data-flow analysis for C and C++ source code that can correctly generate performant data mappings and automatically insert the data directives into OpenMP source code. The tool was evaluated on 9 benchmarks and compared against expert implementations from the Rodinia and HeCBench suites. Our results show a geometric mean speedup of $2.8\times$ over the default implicit data mapping rules and $1.05\times$ speedup over the expert-defined mappings. Across all 9 benchmarks, our tool also demonstrated a substantial reduction of redundant data transfer, with a geometric mean reduction of 2.1GB. These findings highlight the effectiveness of our approach in automatically generating performant data mappings for OpenMP offloading applications. By automating this process, our tool not only enhances the performance of OpenMP offloading applications but also saves programmers significant time and effort, accelerating the development of such applications.

ACKNOWLEDGEMENTS

---

[1] https://researchit.las.iastate.edu/

REFERENCES

[1] "TOP500 List," Nov. 2023. [Online]. Available: https://www.top500.org/lists/top500/2023/11/

[2] *OpenMP API 4.0 Specification*, OpenMP Architecture Review Board, Jul. 2013. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[3] *OpenMP API 5.2 Specification*, OpenMP Architecture Review Board, Nov. 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[4] Clang LibTooling. Accessed: Apr. 1, 2024. [Online]. Available: https://clang.llvm.org/docs/LibTooling.html

[5] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[6] *OpenMP Offloading Tuning Guide*, Intel, Nov. 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2024-0/openmp-offloading-tuning-guide.html

[7] nvprof. Nvidia. Accessed: Sep. 1, 2023. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/

[8] NVIDIA Nsight Systems. Nvidia. Accessed: Apr. 1, 2024. [Online]. Available: https://developer.nvidia.com/nsight-systems

[9] H. Guo, L. Zhang, Y. Zhang, J. Li, X. Xu, L. Liu, K. Cai, D. Wu, S. Yang, L. Kong, and X. Gao, "Openmp offloading data transfer optimization for dcus," *The Journal of Supercomputing*, Aug. 2023.

[10] Z. Jin and J. S. Vetter, "A benchmark suite for improving performance portability of the sycl programming model," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 325–327.

[11] Z. Jin, "The rodinia benchmarks in sycl," Oak Ridge National Laboratory, Tech. Rep., Dec. 2021. [Online]. Available: https://www.osti.gov/biblio/1838961

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[13] M. Yousefi. (2018, Oct.) Allen cahn cuda (phase-field simulation of dendritic solidification). [Online]. Available: https://github.com/myousefi2016/Allen-Cahn-CUDA

[14] T. Mitchell, *Machine Learning*. McGraw Hill, 1997, companion to Chapter 4. [Online]. Available: http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html

[15] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *High Performance Computing – HiPC 2007*, S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 197–208.

[16] J. Stone. (2009, Oct.) An introduction to opencl. Theoretical and Computational Biophysics Group, Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign. [Online]. Available: https://www.ks.uiuc.edu/Research/gpu/files/openclintro.pdf

[17] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotspot: a compact thermal modeling methodology for early-stage vlsi design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 5, pp. 501–513, 2006.

[18] I. Karlin, J. Keasler, and J. R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab, Tech. Rep., Jul. 2013. [Online]. Available: https://www.osti.gov/biblio/1090032

[19] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022283670900574

[20] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[22] J. H. Davis, C. Daley, S. Pophale, T. Huber, S. Chandrasekaran, and N. J. Wright, "Performance assessment of openmp compilers targeting nvidia v100 gpus," in *Accelerator Programming Using Directives*, S. Bhalachandra, S. Wienke, S. Chandrasekaran, and G. Juckeland, Eds. Cham: Springer International Publishing, 2021, pp. 25–44.

[23] J. M. Diaz, K. Friedline, S. Pophale, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran, "Analysis of openmp 4.5 offloading in implementations: Correctness and overhead," *Parallel Computing*, vol. 89, p. 102546, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819119301371

[24] M. Thavappiragasam, W. Elwasif, and A. Sedova, "Portability for gpu-accelerated molecular docking applications for cloud and hpc: can portable compiler directives provide performance across all platforms?" in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 975–984.

[25] Openmp compilers & tools. OpenMP. Accessed: Mar. 31, 2024. [Online]. Available: https://www.openmp.org/resources/openmp-compilers-tools

[26] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[27] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

[28] A. Mishra, A. M. Malik, and B. Chapman, "Data transfer and reuse analysis tool for gpu-offloading using openmp," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, K. Milfeld, B. R. de Supinski, L. Koesterke, and J. Klinkenberg, Eds. Cham: Springer International Publishing, 2020, pp. 280–294.

[29] A. Mishra, A. M. Malik, M. Lin, and B. Chapman, "Openmp advisor: A compiler tool for heterogeneous architectures," in *OpenMP: Advanced Task-Based, Device and Compiler Programming*, S. McIntosh-Smith, M. Klemm, B. R. de Supinski, T. Deakin, and J. Klinkenberg, Eds. Cham: Springer Nature Switzerland, 2023, pp. 34–48.

[30] P. Barua, J. Shirako, W. Tsang, J. Paudel, W. Chen, and V. Sarkar, "Ompsan: Static verification of openmp's data mapping constructs," in *OpenMP: Conquering the Full Hardware Spectrum*, X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, Eds. Cham: Springer International Publishing, 2019, pp. 3–18.

[31] P. Barua, J. Zhao, and V. Sarkar, "Ompmemopt: Optimized memory movement for heterogeneous computing," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds. Cham: Springer International Publishing, 2020, pp. 200–216.