OmpMemOpt: Optimized Memory Movement for Heterogeneous Computing

Prithayan Barua(✉), Jisheng Zhao, and Vivek Sarkar

Georgia Institute of Technology, Atlanta, GA, USA
{prithayan,jzhao367,vsarkar}@gatech.edu

Abstract. The fast development of acceleration architectures and applications has made heterogeneous computing the norm for high-performance computing. The cost of high volume data movement to the accelerators is an important bottleneck both in terms of application performance and developer productivity. Memory management is still a manual task performed tediously by expert programmers. In this paper, we develop a compiler analysis to automate memory management for heterogeneous computing. We propose an optimization framework that casts the problem of detection and removal of redundant data movements into a partial redundancy elimination (PRE) problem and applies the lazy code motion technique to optimize these data movements. We chose OpenMP as the underlying parallel programming model and implemented our optimization framework in the LLVM toolchain. We evaluated it with ten benchmarks and obtained a geometric speedup of 2.3×, and reduced on average 50% of the total bytes transferred between the host and GPU.

Keywords: Compiler optimization · GPUs · OpenMP · Memory management

1 Introduction

As high-performance computing enters an era of extreme heterogeneity, there is an increasing proliferation of general and special purpose accelerators as well as a concerted effort by higher-level parallel programming models to support heterogeneous computing, e.g., OpenMP, OpenACC, X10, Chapel, Julia. Data movement between the host and accelerators is a fundamental operation in heterogeneous computing, and parallel programming models vary in supporting data movement either explicitly or implicitly. Data movement is also a significant source of overhead, both in execution time and energy. Thus, minimizing data movement while maintaining the correctness of a program is one of the most important optimizations that compilers and application developers focus on [1,6,7,11,17].

We propose a program analysis framework to enable the compiler to automatically detect and remove redundant memory copies. We use OpenMP 4.5[1] as

[1] www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

M. Malawski and K. Rzadca (Eds.): Euro-Par 2020, LNCS 12247, pp. 200–216, 2020.
https://doi.org/10.1007/978-3-030-57675-2_13

an example parallel programming model to demonstrate our optimization framework. We can offload a region of code to accelerators like GPUs using OpenMP. An application developer can specify several different kinds and combinations of OpenMP directives to extract optimal performance from specific hardware. But the developer also needs to ensure the correctness and absence of data races while manually optimizing the application. Given the complexity of OpenMP specifications, this is a nontrivial task and requires time-consuming efforts from expert programmers. Tools like OmpSan [4] help developers debug incorrect usage of OpenMP memory mapping directives. Our objective is to investigate how the compiler can optimize the memory management operations, while the user only needs to specify synchronization operations needed for correctness.

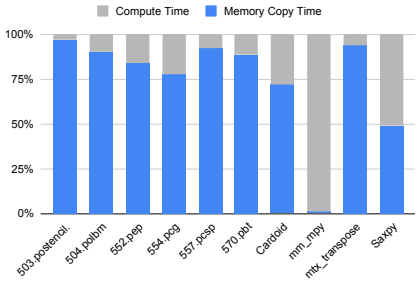


Fig. 1. Compute time vs memory copy

Figure 1 shows the significance of the data movement overhead for 10 OpenMP GPU applications discussed later in Sect. 5. In this experiment, the kernels don't use any explicit memory mapping and rely on the default behavior, which is to copy data from a host to the GPU before launching the kernel and back to the host after it executes. It compares the % time spent on computing vs. data transfer operations. The experiment illustrates the inefficiency of the default mapping since except for the compute-intensive *mm_mpy* and *saxpy* kernels, over 70% of the time is spent on memory transfer operations in the remaining benchmarks. In this paper, we formalize the data movement optimization problem and define an intermediate representation suitable for the analysis of memory accesses and data movements in heterogeneous computing. Then, we introduce our optimization framework hat uses the intermediate representation to perform lazy code motion and partial redundancy elimination on data movement operations.

The main contributions of this paper include:

1. We introduce a general optimization framework to apply partial redundancy elimination, that uses dataflow analysis to identify redundancies in data movement, and a code transformation to eliminate such redundancies.
2. We extend past work on Heap SSA [9] to a new Location-Aware heap SSA (LASSA) to consider heterogeneous memory spaces. We implement construction of LASSA, and its associated optimizations, in the LLVM tool chain.
3. We evaluate our approach using real-world heterogeneous computing applications.

2 Background

2.1 OpenMP Execution Model

In this section, we briefly discuss the OpenMP programming model. We use the term device to refer to a computing resource. The host device is the CPU that begins executing the program. There are optional accelerators like a GPU that are called target devices. An OpenMP program begins as a single thread of sequential execution, called the master thread, which runs on the host device. The OpenMP target directive specifies a block of code to offload to the device. One or more target devices can be available to the host for offloading code and data. The target directive generates a new target task, which may execute on a target device. The target task starts with an initial thread, and teams of threads can be optionally created depending on the usage of team/parallel constructs.

An important aspect of the memory model[2] [10] is that the tasks running on the host and tasks running on the target devices have separate states that are not shared. Each host device and target device has at least one attached storage resource(s) that is private to them. This is called a *memory space* in OpenMP terminology. When the host and target task need to communicate, they do so by explicitly copying data from one memory space to another. The memory space is a persistent resource, e.g., the target memory space retains all data allocated in its space unless it is explicitly deleted.

2.2 Heap SSA Form

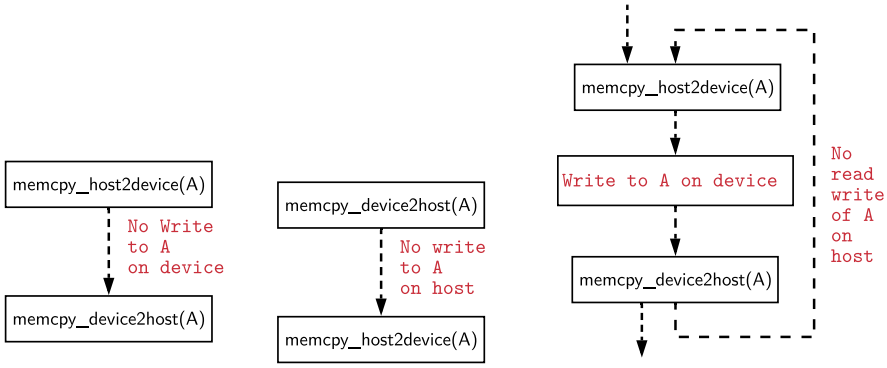
Heap SSA [9] is an intermediate representation that extends Array SSA form [14] to capture reads and writes to heap-allocated data. Heap SSA models each access of a disjoint memory space as a distinct logical "heap array". Heap SSA employs use:$u\phi$ and definition:$d\phi$ operators to chain memory load and store operations, respectively. It was designed for strongly typed languages like Java, but it is also applicable to weakly typed languages by introducing a uniform heap array that captures element-level dataflow information for heap data structures [21].

3 Motivation

Figure 2 shows some typical cases of redundant memory copies that programmers need to detect and optimize manually. Here, *memcpy_host2device* copies an array from host to device, while *memcpy_device2host* copies it back from the device to host. It shows a dummy CFG in which the dotted line represents an arbitrary sequence of code, which respects the condition mentioned alongside it.

Redundancy Pattern 1. Figure 2a is the simplest use case; if a kernel launched on the device does not update an array, then there is no need to copy the array back to the host. The default behavior of OpenMP target constructs is to copy in and out every array.

[2] www.openmp.org/wp-content/uploads/openmp-4.5.pdf.



(a) Redundant Copy-out from device **(b)** Redundant Copy-in to device **(c)** Redundant copies within loop

Fig. 2. Common patterns of redundancy

```
1  int A[10];
2  #pragma omp target map(A)
3  {
4    for (i = 0 ; i < 10; i++)
5      A[i] = i;
6  }
7  print(A)
8  #pragma omp target map(A)
9  {
10   for (i = 0 ; i < 10; i++)
11     A[i] + = i;
12 }
13 print(A)
```

(a) Default memory map

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

(b) Explicitly specify data copies

Fig. 3. Redundancy Pattern 2

Redundancy Pattern 2. Figure 2b shows the second pattern, when a host-to-device copy is redundant since the array is already the latest version on the device because of the persistent device storage. After executing a kernel on the device, we copy the array back from device-to-host. Figure 3a shows this coding pattern using OpenMP target offloading constructs. Line 2 launches a kernel on the device that updates the array A. Then the kernel launched on line 8 reads and updates the array A in the device memory. The print statement on line 7 is executed on the host. It only reads the array, and it is not updated on the host before launching the second kernel. The device already has the latest version of the array on line 8, and thus the copy is redundant. Figure 3b shows the usage of *target data map* clause on line 2 to handle such redundancies. We explicitly leave the array on the device's persistent memory for later use.

```
1 int A[10];
2 for (t = 0 ; t < 100; t++) {
3   #pragma omp target map(A)
4   {
5     for (i = 0 ; i < 10; i++)
6       A[i] += i;
7   }
8 }
9 print(A)
```

(a) Kernel Launch within loop

```
1  int A[10];
2  #pragma omp target data map(tofrom:A)
3  {
4   for (t = 0 ; t < 100; t++) {
5     #pragma omp target map(alloc:A)
6     {
7       for (i = 0 ; i < 10; i++)
8         A[i] += i;
9     }
10  }
11 }
12 print(A)
```

(b) Explicit memory copies

Fig. 4. Redundant copies within loop, Pattern 3

This example motivates our claim that optimizing even simple memory copy redundancies requires nontrivial understanding of OpenMP spec and the knowledge of all the available directives and their possible usage.

Redundancy Pattern 3. Figure 2c shows another pattern where a host loop launches a kernel on the device iteratively. This host loop does a host-to-device copy before launching the kernel and again device-to-host copy after it finishes. Both these copies are redundant since the host does not access the copied memory inside the loop. Figure 4a shows the OpenMP example for the third case, the target construct on line 3 executes host-to-device copy before launching the kernel on the device and then device-to-host copy after the kernel returns. But, since the outer loop of line 2, executing on the host does not access the array, both the copies are loop-invariant. In this case, it is legal to move the host-to-device memory copy before the loop, and the device-to-host memory copy after the outer loop. Figure 4b shows the usage of memory map environments to remove the redundancy.

In this section, we presented three simple examples of redundant memory copies to motivate our work. But these patterns can be generalized to complex real-world use cases. The dotted line of the CFG can denote arbitrarily complex source code. Hence the redundant memory copies can even occur across different function calls and source files. This makes manual detection of redundant memory copies and its optimization much more complicated and error-prone. Several OpenMP application developers have provided similar feedback regarding these issues related to manual optimization of memory management. The common uses cases are usually scientific applications with large legacy codebases, that are being ported to GPUs using the OpenMP *target offloading* feature launched in version *4.5*. The nontrivial effort required for manual memory management is our motivation to develop a compiler optimization to automate removal of such redundant memory copies.

3.1 Challenges

To address the problem introduced above, we need to address the following challenges:

- Representation of concurrent memory accesses to the same array elements;
- Reasoning about the definition-use (def-use) relationships among array accesses across different memory spaces;
- Whole program analysis that infers optimal program points for inserting memory copy operations, and detects redundant data movements.

4 Our Approach

Problem Statement. Based on the programming-model, first, the compiler needs to identify where to insert the memory copy operations to ensure correctness. Then an analysis is required to determine partially and fully redundant memory copies. Finally, a code transformation is needed to remove all the redundancies.

Proposed Solution. We design an intermediate representation to express the memory model of the programming paradigm and develop an analysis based on that representation, to optimize redundant memory copies between different memory spaces. We make the following basic assumptions

- We assume that pointer analysis can disambiguate named arrays. If the alias analysis fails to identify each array uniquely, our optimization fails.
- To keep the analysis simple, any element-level access is conservatively assumed to access the entire array. This constraint can be removed by performing an index range analysis for each array access.

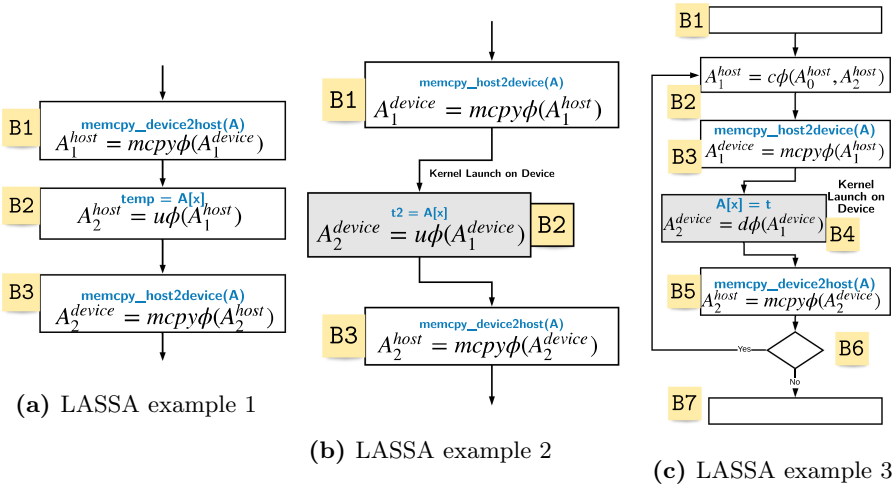
4.1 Location Aware Heap SSA

The heterogeneous computing patterns mainly deal with array-based data structures over one or more memory spaces of different devices. In this section, we introduce the Location-Aware Heap SSA (LASSA) IR, which extends Heap SSA [9] to take into account the memory space in which each array resides. To uniquely identify each array access in a LASSA program, we create a new version of the array for every corresponding access to it. We define LASSA operators that map an array version in one memory space to another array version in the same or different memory space. We call these array versions as a definition.

We use the notation, D_i^r, to denote the i^{th} definition in memory space r.

Definition 1. *We define the following operators in LASSA for an array A:*

1. $A_i^r = d\phi(A_j^r)$ *creates a new definition. such that,* A_j^r *is the prevailing definition of A just prior to* A_i^r *in the memory space r.*
2. $A_k^r = c\phi(A_i^r, A_j^r)$*, creates a control merge of the definitions* $\{A_i^r, A_j^r\}$.
3. $A_i^r = u\phi(A_j^r)$*, denotes the read of A.*



(a) LASSA example 1

(b) LASSA example 2

(c) LASSA example 3

Fig. 5. Example LASSA operators, shaded blocks are executed on device

4. $A_i^r = mcpy\phi(A_j^p)$, *creates a new definition of* A*, due to a copy from memory space* p *to memory space* r*, this is a new operator that was not present in Heap SSA.*

The semantics of the $d\phi$ and $u\phi$ operators are associated with the respective memory write and read operations. The $u\phi$ operator also generates a dummy definition, for array reads. The main purpose of the $u\phi$ operator is to remove redundant copy statements. The control merge operator $c\phi$ merges the reaching definitions from two incoming paths and creates a new definition. The $u\phi$, $d\phi$ and $c\phi$ are the same operators as in Heap SSA [9]. A $mcpy\phi$ is associated with a program point where the memory from source memory space data is flushed/written out to the destination memory space. This guarantees the copied data is visible to any subsequent memory operations. We can use $mcpy\phi$ for both synchronous or asynchronous memory copy. But, the placement of the operator depends only on when the actual write is visible, as defined by the memory concurrency model. For an array A and device memories $dev1, dev2$, We use the notation $A_{dev1,dev2}$ to denote that both the memory spaces $dev1$ and $dev2$ have exactly the same copy of array A. We now discuss some example LASSA representations.

Case 1. Figure 5a shows an example LASSA IR for case 1. Basic Block $B1$ copies data back from the device to the host, assuming there is some preceding kernel that executes on the device not shown here. Assuming A_1^{device} is the most recent version of the array on the device, the copy creates A_1^{host}, a new version of the array on the host represented by $A_1^{host} = mcpy\phi(A_1^{device})$. Next, $B2$ reads a location of the array on the host, represented by the $u\phi$ operator. Finally, $B3$ uses the $mcpy\phi$ operator to denote the host-to-device memory copy.

Case 2. Figure 5b shows the LASSA IR for case 2. $B2$ is a kernel executed on device, denoted by the shaded block in the figure. $B1$ denotes the host-to-device memory copy with the $mcpy\phi$ operator, and it updates the version of the array on device to A_2^{device}. After the copy, A_1^{device} is the updated version of the array on the device read by the $u\phi$ operator of $B2$. $B3$ copies the array back to the host after $B2$ finishes execution on the device.

Case 3. In Figure 5c, $B4$ is a kernel launched on device, which is executed inside a loop. This represents the loop invariant case. $B3$ copies the array from the host-to-device, and $B5$ copies the array back from the device-to-host. $B2$ is the entry block of the loop, it merges the control from the back edge. Assuming A_0^{host} is the last version of array on the host before entry to loop, the $A_1^{host} = c\phi(A_0^{host}, A_2^{host})$ merges the A_2^{host} from loop body to create a new version A_1^{host}. $B4$ updates the array on device, denoted by the $d\phi$ operator which creates the version A_2^{device}, that is copied back to the host at $B5$.

4.2 Redundancy

We will use the data flow analysis defined in Chapter 10 of the compiler textbook [20] for partial redundancy elimination [8,15] of memory copies across different memory spaces. In this section, we define the data flow properties in terms of the $mcpy\phi$ LASSA operator.

Definition 2 ***Availability:*** *An $mcpy\phi$ of A is said to be available between two memory spaces* m *and* p*, at a basic block* B*, if any memory copy of A between* m *and* p *is redundant at* B *since both memory spaces have the same version of the array after the last copy. This is a forward analysis.*

Availability implies, after the last copy: $D_i^m = mcpy\phi(A, D_j^p)$, D_i^m is still the most recent version of the array A on memory space m, and D_j^p is the most recent version of array A on memory space p. It is computed using a forward analysis. Given a basic block B, $AvailOut(B)$ denotes the availability at the exit of B. $DEExpr(B)$ and $UEExpr(B)$ is the set of downward and upward exposed $mcpy\phi$ operators respectively. They are defined in Table 1. $ExprKill(B)$ denotes the memory copies that are killed due to an update. We use the same definition of $AvailOut$ from [20],

$$AvailOut(n) = \bigcap_{m \in preds(n)} (DEExpr(m) \cup (AvailOut(m) \cap \overline{ExprKill(m)}))$$

$AvailOut(inputBlock) = \phi$, and for all other blocks $AvailOut(B) = All\ Copies$,

Definition 3 ***Anticipability:*** *An $mcpy\phi$ of* A *is anticipable (very busy) between memory spaces* m *and* p*, on exit of a basic block* B*, if every path that leaves* B*, executes a memory copy of* A *between* m *and* p*, and it is legal to hoist it to the end of* B.

Table 1. Transfer functions for the basic block local properties

LASSA operators	Downward exposed	Upward exposed	Killed copy
Explanation	if $A_{\{p,q\}} \in DEExpr(B)$ then, the version of A on p and q are same at the end of B	if $A_{\{p,q\}} \in UEExpr(B)$ then, copy from p to q can be hoisted up at the head of B	Killed Copy
Initialization	$DEExpr(B) = \{\}$	$UEExpr(B) = \{\}$	$ExprKill(B) = \{\}$
Analysis direction	Forward	Backward	Forward
$D_i^r = d\phi(A, D_i^r)$	$DEExpr(B) \setminus A_{\{r,x\}} \forall x$	$UEExpr(B) \setminus A_{\{r,x\}} \forall x$	$ExprKill(B) \cup A_{\{r,x\}} \forall x$
$D_i^r = u\phi(A, D_j^r)$	$DEExpr(B)$	$UEExpr(B)$	$ExprKill(B)$
$D_i^r = mcpy\phi(A, D_j^q)$	$DEExpr(B) \cup A_{\{q,r\}}$	$UEExpr(B) \cup A_{\{q,r\}}$	$ExprKill(B)$

Table 2. Computing availability and anticipability

	Available out
Figure 5a $B1$	$A_{\{host,device\}}$
Figure 5a $B2$	$A_{\{host,device\}}$
Figure 5b $B1$	$A_{\{host,device\}}$
Figure 5b $B2$	$A_{\{host,device\}}$

(a) Redundancy

	Available out	Anticipable in
$B1$	ϕ	$A_{\{host,device\}}$
$B2$	ϕ	$A_{\{host,device\}}$
$B3$	$A_{\{host,device\}}$	$A_{\{host,device\}}$
$B4$	ϕ	ϕ
$B5$	$A_{\{host,device\}}$	$A_{\{host,device\}}$
$B6$	$A_{\{host,device\}}$	ϕ
$B7$	$A_{\{host,device\}}$	ϕ

(b) Partial redundancy

Anticipability is computed by a backward analysis using the following equations,

$$AntIn(m) = UEExpr(m) \cup (AntOut(m) \cap \overline{ExprKill(m)})$$

$$AntOut(n) = \bigcap_{m \in succ(n)} AntIn(m), \quad m \neq Exit\ Block$$

$AntOut(Exit\ Block) = \phi$, and for all other blocks $AntOut(n) = All\ Copies$

To compute the availability and anticipability, we define a lattice over the $mcpy\phi$ of array variables. We use $A_{\{src,dst\}}$ to denote that the memory copy of A between src and dst is redundant, that is both memory spaces have exactly the same copy of A. Our analysis is based on the "lazy code motion" data-flow-equations from [8]. Table 1 defines the local properties used to compute the availability and anticipability.

Definition 4 ***Redundancy:*** *A copy statement between memory spaces* m *and* p *for a particular array* A *is redundant, if both the memory spaces already have the same version of* A.

A memory copy, $D_i^p = mcpy\phi(A, D_j^m)$ is redundant if $A_{\{m,p\}} \in AvailOut(D_j^m)$

Example of Redundancy. Consider Fig. 5a and Fig. 5b, in both these cases $B1$ and $B3$ have an $mcpy\phi$ operator, and there is no write to the array between this pair of $mcpy\phi$ statements. Thus, as Table 2a shows, $A_{\{host,device\}}$ is available at the entry to basic block $B3$ which means the host and device memory space have the same copy of the array and any further copy is redundant. Thus we can remove the memory copy from the $B3$ in the first two cases.

Definition 5 ***Partial Redundancy:*** *A copy statement between memory spaces* m *and* p *for a particular array* A*, constitutes a partial redundancy, if both the memory spaces already have an updated copy on some but not all paths reaching the copy statement.*

Example of Partial Redundancy. Consider the loop invariant case in Fig. 5c. As Table 2b shows, The memory copy of $B3$ is anticipable at the entry of both $B1$ and $B2$, that is to the entry block of the loop. But the device definition in $B4$ makes sure that the $B5$ copy is not redundant. Now, the copy of $B5$ is available at the exit of $B5$ and also till the loop exit block $B7$. Consider the two edges of $B1-B2$ and $B6-B2$, $A_{\{host,device\}}$ is available only on the back edge $B6-B2$, but not on the entry to the loop. Hence it is partially redundant at $B2$.

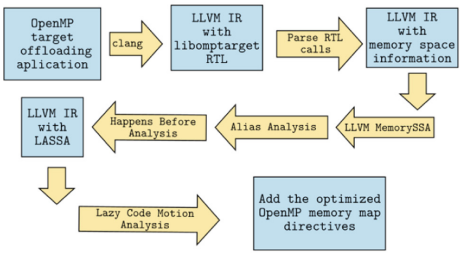
4.3 Lazy Code Motion

Partial redundancy elimination (PRE) [15] eliminates redundant computation of expressions in programs by moving invariant computations out of loops and also eliminating identical computations that are performed more than once on any execution path. In this paper, we use the formulation from [20] and [8]. Our customized PRE algorithm for data movements has the following steps:

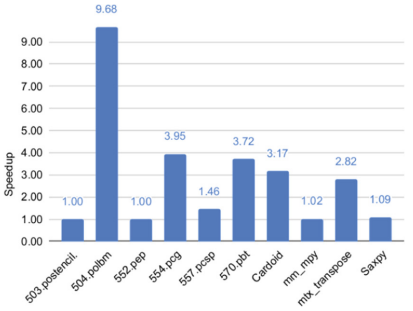
1. *Basic block local properties*: compute the local properties of upward-exposed and downward-exposed $mcpy\phi$ operators using the transfer functions defined over LASSA operators in Table 1.
2. *Solve the data flow equations*: compute available and anticipable copy operations according to Definition 2 and Definition 3.

3. *Determine Earliest and Latest placement*: given the solutions of availability and anticipability, we can determine the earliest point in the program at which it is safe to hoist the copy statement. It is profitable to insert a copy statement at a basic block B, if it makes other copy statements redundant. Again we use the original data flow equations [20], to solve for earliest and later placement.
4. *Redundant copies*: this translates to identifying redundant memory copy statements according to Definition 4.
5. *Code rewrite*: identify the program point to insert the memory copy, and the set of redundant memory copies that can be deleted.

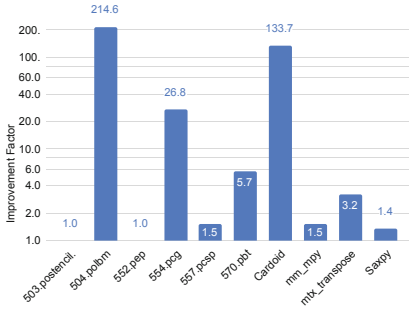
Note that dataflow analysis on the LASSA IR ensures that the transformed program produces the same output as the original output. The semantics of the *mcpyϕ* IR ensures the legality of the optimization.



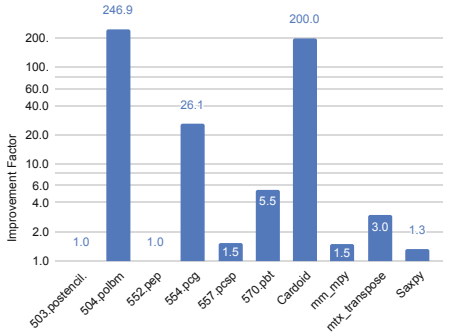
(a) Framework LLVM implementation



(b) Speedup compared to default map



(c) Improvement in Memory Copy Time



(d) Improvement in Total Bytes Copied

Fig. 6. Experimental framework and results

5 Evaluation

Implementation. We implemented our analysis in the LLVM 9.0.1 compiler framework[3]. Figure 6a shows an overview of our analysis and the optimization framework. We used *Clang* to emit LLVM IR using the target-independent "libomptarget" OpenMP offloading library. The analysis pass then analyzes the API calls and their arguments to infer the offload pragmas specified by the user. We implemented an Andersen like flow-insensitive alias analysis, and also used two LLVM built-in analyses: scalar evolution for array index analysis and memory SSA[4] for chaining memory access and data copy operations.

For optimal memory copy insertion, we developed our analysis pass *OmpMemOpt*. It performs an inter-procedural analysis to detect redundant memory copies. Based on the analysis results, we infer the optimal places to insert the OpenMP memory copy constructs. Finally, we developed a Perl script to insert the appropriate memory mapping directives into the input source files. Thus, given an OpenMP target offloading application with no explicit memory management, our tool analyzes the program and finally generates the modified source files after adding the optimal set of OpenMP memory map directives.

Experimental Setup. We use the OpenMP benchmarks from SPEC ACCEL v1.2 to evaluate our analysis and optimizations. We exclude Fortran applications from our evaluation, since they are not supported by our current tool chain; we also exclude benchmarks that do not use target offloading. We show results for the 6 SPEC benchmarks, and also include 4 other applications: *saxpy*, *Cardoid* ,*Matrix Multiply* and *Matrix Transpose*.

Our experimental results were obtained from a Linux (Ubuntu 18.04.3) workstation, Intel Core i5-7600 CPU (3.50GHz), 16GB memory and an Nvidia "TITAN Xp" GPU with 12GB memory and CUDA 10.1.

Experimental Result and Discussion. We removed all the explicit memory mapping constructs specified in the benchmarks to obtain our baseline. The host performs host-to-device copy in the baseline version before launching every kernel on the device and device-to-host copy after the kernel finishes execution.

After running our optimization on the benchmark, we have three versions of each application: the baseline, *OmpMemOpt* optimized version, and the original hand-optimized benchmark. We compare the performance of these three versions to evaluate our framework. We measure the efficiency on such metrics: the improvement of execution time, the reducing of data volumes and time consumed on data movement.

[3] http://llvm.org/.
[4] https://llvm.org/docs/MemorySSA.html.

We did the following study for the comparison with baseline code. Figure 6b shows the overall speedup obtained by our approach compared to the naive data mapping baseline. As we can see except *503* and *552*, all the benchmarks show a speedup ranging from 1.02× to almost 10×. The 503 and 552 benchmarks did not get a chance to be optimized due to limitations in the precision of the alias analysis used—flow-insensitive pointer analysis could not disambiguate the array references in those two benchmarks.

Table 3. Comparison of our achieved speedup with manually optimized speedup

Benchmark	In put	Memory copy time	Total time	Manual speedup	Our speedup
503.postencil	ref	954491.4	983668.3	33.5	1.0
503.postencil	test	3108.6	3205.2	25.6	1.0
503.postencil	train	3116.8	3211.5	25.5	1.0
504.polbm	ref	497859.3	553697.4	9.5	9.5
504.polbm	test	2014.5	2243.0	7.0	7.0
504.polbm	train	30222.4	33615.8	9.4	9.3
552.pep	ref	563182.7	671546.9	5.7	1.0
552.pep	test	469.8	653.9	3.6	1.0
552.pep	train	35889.8	42726.6	5.8	1.0
554.pcg	ref	807757.1	1040824.3	4.5	3.9
554.pcg	test	24129.3	31056.1	4.4	4.0
554.pcg	train	88261.0	113651.9	4.5	4.0
557.pcsp	ref	1204141.9	1308006.4	2.0	1.5
557.pcsp	test	20098.1	20229.1	1.5	1.5
557.pcsp	train	464849.7	475782.2	2.0	1.5
570.pbt	ref	3750608.5	4221773.7	3.7	3.7
570.pbt	test	1321807.1	1339861.0	2.6	2.6
570.pbt	train	2563893.7	2728456.2	3.6	3.6
Cardoid		838.5	1163.8	3.17	3.17
mm_mpy		750.8	54555	1.02	1.02
mtx_transpose		16.66	17.7	2.8	2.8
Saxpy		154.7	315.9	1.09	1.09

Figure 6c explains the reason for the speedup, by showing the improvement factor of memory copy time, compared to the baseline. A significant point to note here is that the performance gain is mostly dependent on the problem size (i.e., input data size). This also implies that the efficiency depends on the data volume reduced for transfer.

Finally, Fig. 6d gives a quantization study of the data volume transferred between the host and the device. It shows the reduction in total bytes copied. As is evident, there is a correlation between the factor by which total bytes are reduced and the obtained speedup. The speedup also depends on the pattern of computation. As the *Matrix Multiplication* example shows, even though there is a 1.5× reduction of memory-copy-time, it does not result in a speedup since the application is compute-intensive. In the benchmark *Cardoid*, there is an outer loop which iterates for 100 iterations, and launches an inner loop on the target device. The default semantics of the **target** construct would perform host-to-device and device-to-host copy in each iteration. But, since there is no host access, there is no need to copy the data back and forth every time. This is why almost 100% of the memory copies are eliminated after our optimization. Benchmark *Saxpy* is similar to *Cardoid*, there is an outer loop that launches the target task every iteration, and redundantly copies the data in every iteration.

Table 3 shows the comparison of speedup obtained from our approach with the manually optimized version. The manually optimized version is the original source released as the SPEC ACCEL benchmarks. The 3rd and 4th columns give

the memory copy time and total execution time for baseline code (i.e. naive memory mapping version). The 5th column shows the speedup obtained by comparing user manually optimized code against baseline. And the last column shows the speedup got from our approach. In general, the user manually optimized version gives the better improvement by comparing the last two columns, and our approach (i.e. compiler optimization) got similar performance on *504.polbm* and *570.pbt*. As mentioned above, there is no improvement from *503.postencil* and 552.pep due to the precision issues from pointer alias analysis. This study shows the compiler's potential to automatically generate as efficient code as a programmer's manually optimized version.

6 Related Work

The problem of code generation and communication optimization for distributed memory machines is a classical problem, studied for a long time. Amarasinghe and Lam [1] introduced a data flow analysis framework to generate remote message read/write code, and then detect and remove redundancies in homogeneous distributed computing. Chavarria and Mellor-Crummey [6] proposed a communication coalescing optimization to reduce redundant data transfer for high-performance Fortran applications. Dathathri et al. [7] introduced a polyhedral model to enable static analysis and automatically generate efficient data movement code for non-shared address spaces.

Load elimination and partial code motion are the classic optimizations for eliminating redundant memory loads in a sequential program. In [5], Bodik et al. phrased the load-reuse problem as a path-sensitive analysis problem on the dataflow graph. Their algorithm can detect the reuse pattern for both scalar variable and pointer-based memory load operations. Recently, GPU based heterogeneous computing is becoming the mainstream configuration of high-performance computing. Several compiler optimizations and runtime techniques have been developed for reducing the communication overhead. In [12], Jablin et al. introduced a CPU-GPU Communication Manager (CGCM), which employs a static analysis with a runtime library to optimize CPU-GPU communication. Ramashekar and Bondhugula introduced BBMM [19] for communication optimizations on a multi-GPU system. They applied communication optimization for the tiled loop nest and generated the OpenCL code that uses BBMM runtime API to perform buffer management and data communication.

Ashcraft et al. built a compilation technique [3] that performs whole-program analysis to make the optimal placement of data transfer operations. Their approach is based on a liveness analysis to identify the preliminary scheduling locations for the data transfer and then use the dominator tree to optimize the locations. In [16], Mendonca et al. developed an automatic annotation mechanism for enabling GPU based data parallelism from the source code and eliminate the redundant CPU-GPU data copies.

There are also several runtime based communication optimization techniques for eliminating the CPU-GPU redundant memory copies. Asai et al. [2] discussed

a runtime based approach using data dependence analysis for reducing the memory copy operations in a GPU-enabled version of the Apache Spark framework. Kim et al. developed a runtime communication optimization: Unnecessary Data Transfer Elimination (UDTE) [13], which uses a page-fault mechanism to avoid redundant CPU-GPU memory copies.

Compared with past work, our approach introduced a general compiler optimization framework that optimizes data movement across different memory spaces in heterogeneous computing. The related work mentioned above addressed this problem using runtime based mechanisms. Our framework reduces data movement overheads, and is applicable to parallel programming models that support heterogeneous computation.

7 Conclusion

In this work, we addressed the problem of optimizing data movement across different computation devices in a heterogeneous computing application. Given that many parallel programming language models (e.g., OpenMP, OpenACC) support offloading of computations and data to different accelerators, automatic elimination of redundant memory copies to improve performance, while still ensuring correctness, is an important challenge for compilers. To address this problem, we developed an optimization framework to identify redundant data movements and perform code transformations to eliminate those redundancies. We first extended Heap SSA to a Location-Aware heap SSA form (LASSA), an intermediate representation that can track host-to-device memory copies across multiple devices. Then, we performed a partial redundancy elimination dataflow analysis on LASSA to address the problem of removing redundant data transfers. We evaluated our technique on 10 benchmarks written in OpenMP 4.5 with target offloading constructs. Our approach demonstrated a geometric mean speedup of 2.3× and saved a geometric mean of 3.48 GB in redundant data transfers. For one of our future work directions, we plan to explore the use of immutability information [18] to further reduce the data transfers performed.

References

1. Amarasinghe, S.P., Lam, M.S.: Communication optimization and code generation for distributed memory machines. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI 1993, pp. 126–138. Association for Computing Machinery, New York (1993). https://doi.org/10.1145/155090.155102
2. Asai, R., Okita, M., Ino, F., Hagihara, K.: Transparent avoidance of redundant data transfer on GPU-enabled apache spark. In: Kaeli, D.R., Cavazos, J. (eds.) 11th Workshop on General Purpose Processing using GPUs, GPGPU@PPoPP 2018, Vosendorf (Vienna), Austria, 25 February 2018, pp. 22–30. ACM (2018). https://doi.org/10.1145/3180270.3180276

3. Ashcraft, M.B., Lemon, A., Penry, D.A., Snell, Q.: Compiler optimization of accelerator data transfers. Int. J. Parallel Program. **47**(1), 39–58 (2017). https://doi.org/10.1007/s10766-017-0549-3
4. Barua, P., Shirako, J., Tsang, W., Paudel, J., Chen, W., Sarkar, V.: OMPSan: static verification of openmp's data mapping constructs. In: IWOMP (2019). https://doi.org/10.1007/978-3-030-28596-8_1
5. Bodík, R., Gupta, R., Soffa, M.L.: Load-reuse analysis: design and evaluation. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI 1999, pp. 64–76, Association for Computing Machinery, New York (1999). https://doi.org/10.1145/301618.301643
6. Chavarría-Miranda, D., Mellor-Crummey, J.: Effective communication coalescing for data-parallel applications. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2005, pp. 14–25. Association for Computing Machinery, New York (2005). https://doi.org/10.1145/1065944.1065948
7. Dathathri, R., Reddy, C., Ramashekar, T., Bondhugula, U.: Generating efficient data movement code for heterogeneous architectures with distributed-memory. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT 2013, pp. 375–386. IEEE Press (2013)
8. Drechsler, K.H., Stadel, M.P.: A variation of knoop, rüthing, and steffen's lazy code motion. SIGPLAN Not. **28**(5), 29–38 (1993). https://doi.org/10.1145/152819.152823
9. Fink, S., Knobe, K., Sarkar, V.: Unified analysis of array and object references in strongly typed languages. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 155–174. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-45099-3_9
10. Hoeflinger, J.P., de Supinski, B.R.: The OpenMP memory model. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP -2005. LNCS, vol. 4315, pp. 167–177. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68555-5_14
11. 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 2012, Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2259016.2259038
12. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, 4–8 June 2011, pp. 142–151. ACM (2011). https://doi.org/10.1145/1993498.1993516
13. Kim, J., Lee, Y., Park, J., Lee, J.: Translating openMP device constructs to openCL using unnecessary data transfer elimination. In: West, J., Pancake, C.M. (eds.) Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, 13–18 November 2016, pp. 597–608. IEEE Computer Society (2016). https://doi.org/10.1109/SC.2016.50
14. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998, pp. 107–120. Association for Computing Machinery, New York (1998). https://doi.org/10.1145/268946.268956

15. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, PLDI 1992, pp. 224–234. Association for Computing Machinery, New York (1992). https://doi.org/10.1145/143095.143136
16. Mendonca, G.S.D., Guimarães, B.C.F., Alves, P., Pereira, M.M., Araujo, G., Pereira, F.M.Q.: DawnCC: automatic annotation for data parallelism and offloading. TACO **14**(2), 13:1–13:25 (2017). https://doi.org/10.1145/3084540
17. Pai, S., Govindarajan, R., Thazhuthaveetil, M.J.: Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. PACT 2012, pp. 33–42. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2370816.2370824
18. Pechtchanski, I., Sarkar, V.: Immutability specification and its applications. Concurr. Comput.: Pract. Experience **17**(5–6), 639–662 (2005)
19. Ramashekar, T., Bondhugula, U.: Automatic data allocation and buffer management for multi-GPU machines. ACM Trans. Archit. Code Optim. **10**(4), 1–26 (2013). https://doi.org/10.1145/2544100
20. Torczon, L., Cooper, K.: Engineering A Compiler, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco (2007)
21. Zhao, J., Burke, M.G., Sarkar, V.: Parallel sparse flow-sensitive points-to analysis. In: Proceedings of the 27th International Conference on Compiler Construction, CC 2018, Vienna, Austria, 24–25 February 2018, pp. 59–70. ACM (2018)