Data Transfer and Reuse Analysis Tool for GPU-Offloading Using OpenMP

Alok Mishra[1(✉)], Abid M. Malik[2], and Barbara Chapman[1,2]

[1] Stony Brook University, Stony Brook, NY 11794, USA
{alok.mishra,barbara.chapman}@stonybrook.edu
[2] Brookhaven National Laboratory, Upton, NY 11973, USA
{amalik,bchapman}@bnl.gov

Abstract. In the high performance computing sector, researchers and application developers expend considerable effort to port their applications to GPU-based clusters in order to take advantage of the massive parallelism and energy efficiency of a GPU. Unfortunately porting or writing an application for accelerators, such as GPUs, requires extensive knowledge of the underlying architectures, the application/algorithm and the interfacing programming model, such as CUDA, HIP or OpenMP. Compared to native GPU programming models, OpenMP has a shorter learning curve, is portable and potentially also performance portable. To reduce the developer effort, OpenMP provides implicit data transfer between CPU and GPU. OpenMP users may control the duration of a data object's allocation on the GPU via the use of target data regions, but they do not need to. Unfortunately, unless data mappings are explicitly provided by the user, compilers like Clang move all data accessed by a kernel to the GPU without considering its prior availability on the device. As a result, applications may spend a significant portion of their execution time on data transfer. Yet exploiting data reuse opportunities in an application has the potential to significantly reduce the overall execution time. In this paper we present a source-to-source tool that automatically identifies data in an OpenMP program which do not need to be transferred between CPU and GPU. The tool capitalizes on any data reuse opportunities to insert the pertinent, optimized OpenMP `target data` directives. Our experimental results show considerable reduction in the overall execution time of a set of micro-benchmarks and some benchmark applications from the Rodinia benchmark suite. To the best of our knowledge, no other tool optimizes OpenMP data mappings by identifying and exploiting data reuse opportunities between kernels.

Keywords: Compiler optimization · GPU · Offloading · Compiler · HPC · OpenMP · Clang · LLVM · Data reuse · Data transfer

1 Introduction

GPUs are well known for their massively parallel architectures, as well as exceptional performance and energy efficiency for suitable codes. Supercomputing clusters like Summit [31] derive the lion's share of their compute power from GPUs. Each of the 4,608 nodes of Summit is configured with 2 IBM POWER9 processors and 6 NVIDIA Tesla V100 GPUs. For the last two decades, GPUs are the preferred accelerator in the high performance computing (HPC) sector, where they serve as a co-processor to accelerate general-purpose scientific and engineering application codes. Today, they expedite computational workloads in cutting-edge scientific research in diverse areas such as Physics, Bioinformatics, Chemistry, Climate Modeling, Machine Learning, and much more.

In the HPC context, GPUs are generally considered to be auxiliary processors that are attached to a CPU. A block of code which will run on the device is called a kernel[1]. Since such a kernel often contains critical computations, most application developers expend considerable time and effort to optimize it. Yet both CPU and GPU have their own separate memories, and data must be transferred between them. Orchestrating data motion between the CPU and GPU memories is of vital importance since data transfer is expensive and can often become a major bottleneck in GPU computing. Efficiently managing data transfers is moreover quite burdensome for the application developer. This is the issue we address in our work.

1.1 GPU Offloading Using OpenMP

OpenMP [9] is a directive-based application programming interface that can be used in a Fortran, C or C++ application code to create a parallel program. It is designed for portability, enjoys wide vendor support, and has a much smaller learning curve than native programming models, like CUDA [25] and HiP [3]. From version 4.0 onward, OpenMP supports accelerator devices in a host-device model, for which it offers "*target offloading*" features. The specification provides several options to allow its users to control the duration of a data object's allocation on the GPU, via the use of `target data` directives. Compared to other directive-based methods like OpenACC [32], OpenMP has a broader user community and is more widely available. Features for offloading to devices are under active improvement, as are the implementations [27] in numerous compilers, including Clang/LLVM [21], GCC [12], Intel [17], and Cray [8].

An advantage that OpenMP provides over most native GPU programming models is that it enables implicit data transfer for GPU kernels. Users can choose whether or not to handle data transfer explicitly; in the latter case, the OpenMP compiler will manage the data motion. Unfortunately, compiler support for implicit data transfer is in need of improvement.

1.2 The Problem

All compilers that we had access to and which support OpenMP GPU offloading, like Clang and GCC, handle an implicit data transfer by moving all data needed for kernel computation without considering its availability on the device.

[1] In this paper the term kernel is always used in reference to a GPU kernel.

Thus, all data used in the kernel are transferred both to and from the device at the start and end of the kernel's execution, respectively. In consequence, applications may spend a significant portion of their execution time on data transfer itself. Automatically determining which data does not need to be transferred to the GPU (because it is already there) or from the GPU (when it has not been changed or is no longer required) could lead to better performance by reducing the amount of data transferred and hence the overall execution time. Such an optimization could largely avoid the performance penalty currently associated with the implicit transfer approach and ultimately help increase the fraction of codes that can utilize a GPU by reducing the developer effort. Although there are tools available which perform source-to-source transformations for GPU offloading [22,28], to the best of our knowledge no such tool exists that identifies and exploits opportunities for data reuse between kernels.

1.3 Our Solution

We have developed a Clang-based tool to perform static data reuse analysis between kernels. We selected Clang because it is the front end for the LLVM framework which is now a building block of all major compiler frameworks. The target applications of this tool are those which already use OpenMP for offloading computation to GPU. This tool performs the following actions:

- It identifies all kernels in an application that uses OpenMP for GPU offloading.
- It identifies the data that needs to be transferred between the CPU and GPU for each kernel.
- It automatically recognizes data reuse opportunities between multiple kernels.
- It inserts the pertinent OpenMP target data directives into the original source code to precisely manage data transfers between the host and the device.

Currently, our tool considers traditional data management and not data management through unified memory [20,24]. Optimizing GPU data management through unified memory is planned as future work. We also currently assume that all CPU-GPU data transfers for an input application code is handled by our tool, therefore, there are no explicit data transfers.

The rest of this paper is organized as follows: Sect. 2 provides motivating examples to describe common scenarios in user code which can benefit from data reuse. Section 3 gives a detailed explanation of our strategy for automatically generating code to exploit data reuse opportunities between kernels. Section 4 describes the experimental setup for our research. Section 5 provides a detailed analysis of the results from our experiments using the tool. Section 6 looks at related work and discussion about probable usage of the tool and we conclude in Sect. 7. Future work and planned extensions are discussed in Sect. 8.

2 Motivating Examples

For this work, we analyzed the Rodinia benchmark suite [4] to find example codes where data can be reused on a GPU. The use cases are defined in Sect. 4.

To motivate the utility of our tool, we discuss here two common scenarios in a user code which can profit from data reuse:

Loops: If a kernel is called from within a loop, there is a high probability that data is reused in multiple calls to that kernel. Any data used inside these kernels are potential candidates for reuse. Our tool analyzes data in all such kernels to decide how to efficiently transfer data between the CPU and GPU. As can be seen in Code 1.1, Kernel 1 is called within a while loop and Array_A is reused by every call of the kernel. Similarly Kernel 2 is called within a for loop, and Array_B is reused in every subsequent call to the kernel.

Close Proximity: We define two kernels to be in close proximity to each other if they are both called from the same function. In such a case there is also a high possibility of data reuse between the kernels. As can be seen in Code 1.2, 3 kernels are called inside function `func1`, and the array Array_A is used inside all 3 kernels in different ways. All kernels called within a loop are, by default, considered to be in close proximity to their subsequent calls. Our tool has the ability to detect data reuse in two kernels if they are in close proximity.

```
while(iter < MAX_ITER) {
    // Kernel 1
    #pragma omp target teams distribute parallel for
    for(int i=0; i<N; i++)
        // Compute on Array_A;
    iter++;
}
for(iter = 0; iter < MAX_ITER; iter++) {
    // Kernel 2
    #pragma omp target teams distribute parallel for
    for(int i=0; i<N; i++)
        // Compute on Array_B;
}
```

Code 1.1. Code snippet for kernels called from inside loops

```
void func1 (Array_A) {
    // Kernel 1
    #pragma omp target ...
        // Assigning Array_A

    // Kernel 2
    #pragma omp target ...
        // Updating Array_A

    // Kernel 3
    #pragma omp target ...
        // Using Array_A
}
```

Code 1.2. Code snippet for proximity of kernels

3 Data Reuse Optimization

In this section, we outline the key steps of our approach.

3.1 Problem with OpenMP Implicit Data Transfer

When an OpenMP program begins, an implicit target data region for each device surrounds the whole program [2]. Each device has a device data environment that is defined by its implicit target data region. Any declare target directives, and directives that accept data-mapping attribute clauses, determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment. If a user chooses not to map any data to the device explicitly, then the data is implicitly managed by the compiler. Compilers, such as Clang, identify all variables used in a `target` region and move any data associated with them to the device. The compilers we studied do not take into consideration whether the data is already available on the device. Once the kernel execution is over, all array data is moved back to the host, irrespective of whether or not the data was updated on the kernel or needed beyond the kernel.

3.2 Our Approach

Our tool automatically identifies the data which need to be moved between the host and the device for each OpenMP kernel, and then searches for any data reuse opportunity between these kernels. It then inserts pertinent `target data` directives, which complies with OpenMP Specification 5.0 [7], to precisely manage data transfers between CPU and GPU. The goal of our tool is to modify the original source code (C/C++) by inserting explicit data transfers between the CPU and GPU, optimized to avoid any unnecessary data motion between them. Two advantages of this approach are:

– The user can accept, modify or reject the changes introduced by our tool.
– The updated code can be compiled using any compiler with OpenMP GPU offloading support.

3.3 Implementation

We implemented our framework using Clang/LLVM version 8.0.0 [5]. During the design stage for the tool, we had to decide whether to apply our analysis and optimization in Clang or on the LLVM IR. Yet the LLVM IR is relatively low-level and generally unsuitable for any approach that involves modification to source code. Once LLVM IR has been generated and optimizations are applied to it, it would be quite difficult to pinpoint the location of the source code where we need to insert directives. Thus, we decided to apply our analysis using the Clang Abstract Syntax Tree (AST) [19].

Table 1. Nodes identified as kernels

Clang AST node	OpenMP directive
OMPTargetDirective	omp target
OMPTargetParallelDirective	omp target parallel
OMPTargetParallelForDirective	omp target parallel for
OMPTargetParallelForSimdDirective	omp target parallel for simd
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 simd

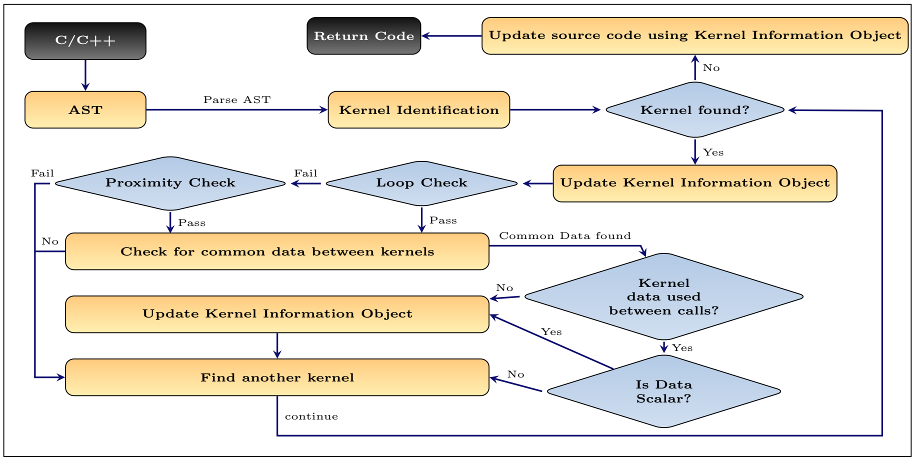

C/C++
Return Code
Update source code using Kernel Information Object
No
AST
Parse AST
Kernel Identification
Kernel found?
Yes
Update Kernel Information Object
Fail
Proximity Check
Fail
Loop Check
Pass
Pass
No
Check for common data between kernels
Common Data found
Kernel data used between calls?
No
Update Kernel Information Object
Yes
Yes
Find another kernel
No
Is Data Scalar?
continue

Fig. 1. Workflow for identifying data transfer opportunities and data reuse in an application using OpenMP for GPU offloading

Clang supports source to source translation, via libTooling [6], even though it is not primarily used for this purpose. In the Clang/LLVM framework [19] most of the analysis is performed on the LLVM Intermediate Representation (IR) and not on the AST. As a result, we had to re-implement some standard analyses such as live variable analysis, data flow graph and code transformation at the AST level for our tool. Consequently, we parse the AST to collect all required information related to the kernels and data variables used in them.

Figure 1 illustrates how our implementation in Clang accurately identifies data reuse opportunity between kernels. First, we parse the AST and identify the kernels in the application. To achieve this, we search for all nodes in the AST as specified in Table 1. For this, we have defined our own Kernel Information Object, which contains information about each identified kernel, e.g., a unique id assigned to the kernel, start and end location of the kernel, function from which the kernel is called, data used inside the kernel, etc. We subsequently use this class to identify variables accessed inside the kernels. The variables are classified into five groups as shown in Table 2 and stored in the Kernel Information Object, to be used during "common data" analysis and for generating the source code.

For each kernel, we implement live variable analysis using the Clang AST, focusing only on variables used inside a kernel. While traversing the AST, we store information about the source code location related to all the variables declared and accessed. Next, we check whether the kernel is called from within a loop and also check for proximity to other kernels, as defined in Sect. 2. Once kernels are identified to be in close proximity to each other, we analyze them to find common data which can be reused across multiple kernels. We use pattern matching on all variables accessed inside the sets of kernels that are in close

Table 2. Types of variables for live variable analysis.

Data types	Description
alloc	These are variables *assigned inside* the kernel for the first time. Data which falls under this category need not be transferred from the host to the device. During code generation these data are mapped with the map type "`alloc`"
to	These are variables *assigned before* but *accessed* inside the kernel. Data which falls under this category need to be transferred from the host to the device. During code generation these data are mapped with the map type "`to`"
from	These are variables that are *updated inside* the kernel and *accessed after* the kernel call. Data which falls under this category needs to be transferred from the device to the host. During code generation these data are mapped with the map type "`from`"
tofrom	These are variables that are *assigned before* a kernel call, *updated inside* it and *accessed after* the kernel execution is complete. Data which falls under this category need to be transferred both ways between the host and the device. During code generation these data are mapped with the map type "`tofrom`"
private	Finally, we have variables which are defined and used *only inside* the kernel. Data which falls under this category does not need to be transferred between the host and the device. During code generation these data are not mapped

proximity to each other, to check for potential data reuse. After identifying common data between kernels, we update the Kernel Information Object. Finally, we use the results of our analysis to update the original source code, inserting the pertinent `target data map` directives to transfer data between the host and the device explicitly.

4 Experimental Setup

To evaluate our benchmarks, we used the SeaWulf computational cluster at Stony Brook University [30]. We ran our tool on four microbenchmarks (Table 3) and six benchmark applications (Table 4) defined in the Rodinia benchmark suite [4]. We selected only those benchmarks from the Rodinia suite, which has GPU offloading support with OpenMP and more than one kernel as part of the code. We modified these benchmarks by removing all data transfer directives so that our tool could introduce new code for them automatically. For each application, we compared two versions:

– **Base Code:** – This is the basic code with implicit data transfer. It does not contain any explicit data transfers.

Table 3. Micro-benchmarks used in the experimentation.

Benchmarks	Description
Three Matrix Multiplication (3 mm)	This is the most basic implementation of multiplying three large matrices. This is a benchmark where two kernels are reusing same data. The experiment used matrices of size 5000×5000 each
Gauss Seidel Method (gauss)	The method for solving linear equations is an iterative method, in which the values for the given variables keep changing until a certain threshold of variance is reached. The experiment used a matrix of size $2^{13} \times 2^{13}$
Laplace Equation (laplace)	The equation in two dimensions with finite differences using jacobi iteration. The experiment used a matrix of size 2000×2000
Single-Precision A·X Plus Y (saxpy)	SAXPY is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. In its simplest form this is a benchmark where two kernels are reusing same data. The experiment used two vectors of size 2^{27} each

Table 4. Updated benchmarks from the Rodinia benchmark suite

Application	Description
Breadth First Search (bfs) [15]	**Graph Algorithm** domain. This benchmark provides the GPU implementations of BFS algorithm which traverses all the connected components in a graph
Hotspot [16]	**Physics Simulation** domain. We re-implemented the transient differential equation solver from HotSpot using target offloading directives for GPU
k-Nearest Neighbor (knn) [11]	**Data Mining** domain. In the implementation it finds the k-nearest neighbors from an unstructured data set
LU Decomposition (lud)	**Linear Algebra** domain. This benchmark is a good example where multiple kernels care called from within a loop and some data shared by these kernels are also used on the host
Needleman Wunsch (nw) [29]	**Bioinformatics** domain. Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments
Particle Filter (p-filter) [14]	**Medical Imaging** domain. This particular implementation is optimized for tracking cells, particularly leukocytes and myocardial cells

– **Optimized Code:** – This is the corresponding code in which our data reuse optimization has been applied to generate explicit data transfers.

We ran the two versions of a benchmark 10 times each and collected information on the amount of data transferred and the total execution time. In its current version, OpenMP in Clang uses CUDA to implement GPU offloading. Our tool is based on Clang/LLVM version 8.0, using OpenMP offloading with CUDA 10.0 in the backend. Therefore, during our experiments, we also tracked how many times the following CUDA APIs related to data transfer are invoked:

– ***cuMemAlloc*** - Allocates bytes of linear memory on the device and returns a pointer to the allocated memory.
– ***cuMemFree*** - Frees the memory space which must have been returned by a previous call to cuMemAlloc.
– ***cuMemcpyHtoD*** - Synchronous copies the specified amount of data from host memory to device memory.
– ***cuMemcpyDtoH*** - Synchronously copies the specified amount of data from device memory to host memory.

We ran these experiments on an NVIDIA Tesla V100 [26] GPU using a PCI-e connector between the CPU and GPU.

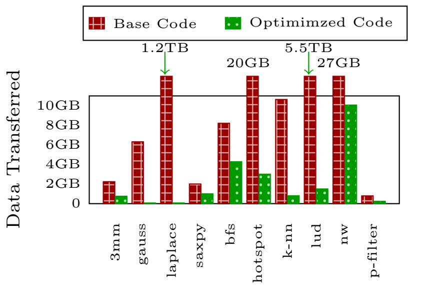
```
#pragma omp target data map(alloc:temp[0:N][0:N])
{ ⟵——— data reuse region starts

#pragma omp target data map(to:A[0:N][0:N],B[0:N][0:N])
#pragma omp target teams distribute parallel for collapse(2)
    for(int i=0; i<N; i++) {
        for(int j=0; j<N; j++) {
            temp[i][j] = 0;
            for(int k=0; k<N; k++)
                temp[i][j] += A[i][k]*B[k][j];
        }
    }
#pragma omp target data map(to:C[0:N][0:N]) map(from:D[0:N][0:N])
#pragma omp target teams distribute parallel for collapse(2)
    for(int i=0; i<N; i++) {
        for(int j=0; j<N; j++) {
            D[i][j] = 0;
            for(int k=0; k<N; k++)
                D[i][j] += temp[i][k]*C[k][j];
        }
    }
} ⟵——— // data reuse region ends
```

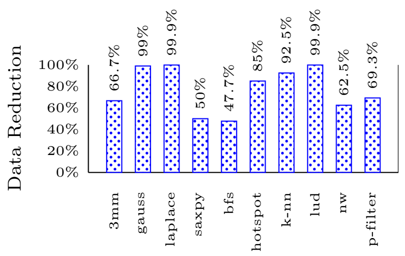
Code 1.3. Example of multiplying three matrices reusing data. Code shown in red is generated automatically by our tool.

5 Results and Analysis

Code 1.3 gives a sample output of our tool when applied to a benchmark code multiplying three matrices. Here the code marked in red is auto-generated by our tool. In this particular example, two kernels use, and reuse data from the



Data transfer between CPU and GPU



% Reduction in size of data transferred when reusing data on GPU

Fig. 2. Comparison of data transferred between CPU and GPU for different benchmarks

array *temp*. Arrays A, B and C only need to be transferred to the device. Since they are not updated on the GPU, we do not need to transfer their values back to the host. Array *temp* is needed only on the GPU, while array D needs to be returned to the host. The array *temp* is assigned on the GPU, so we do not need to transfer its data from the host to the device. We ran the base codes and optimized codes and collected the amount of data transferred between the host and device. We determined that more than 2 GB of data was transferred between the host and device in the base case.

In contrast, in our optimized code, only 763 MB of data was transferred. As can be observed in Fig. 2, there is a reduction of 66.67% in data transfer, accomplished by automatically adding three lines of OpenMP *target data* directives to manage data transfer (cf. Code 1.3). After running our tool on all the benchmark applications, we collected the amount of data transferred for the base and optimized code for each of them, and found that in all cases, less data was transferred in the optimized code than in the base code. This can clearly be observed in Fig. 2.

For Rodinia's *LU-Decomposition* benchmark, in the base case 5.5 TB (TeraByte!) of data was transferred between the host and the device, as compared to 1.5 GB in the optimized code. This is a huge reduction of 99.97%. Also, the *Laplace Equation* micro-benchmark transferred 1.2 TB of data in the base case, in comparison to only 61 MB in the optimized code (99.99% reduction). As evident from Fig. 2(b), we observed a tremendous reduction in the amount of data transferred between the host and device, with BFS being the lowest at 47.7%.

We then took a closer look at the number of times the CUDA data transfer APIs are called for each application. As shown in Fig. 3(C), *Laplace Equation* called each of the four APIs – `cuMemAlloc`, `cuMemFree`, `cuMemcpyHtoD` and `cuMemcpyDtoH` around 25000 times in its base case. Upon further analysis of the base code, we discovered that it was calling a kernel from inside a loop which iterated for 5000 times. Of the four arrays and one variable which were used inside the kernel, only the variable was used both on the host and the device. In our optimized code, the data transfer for the arrays was moved outside the

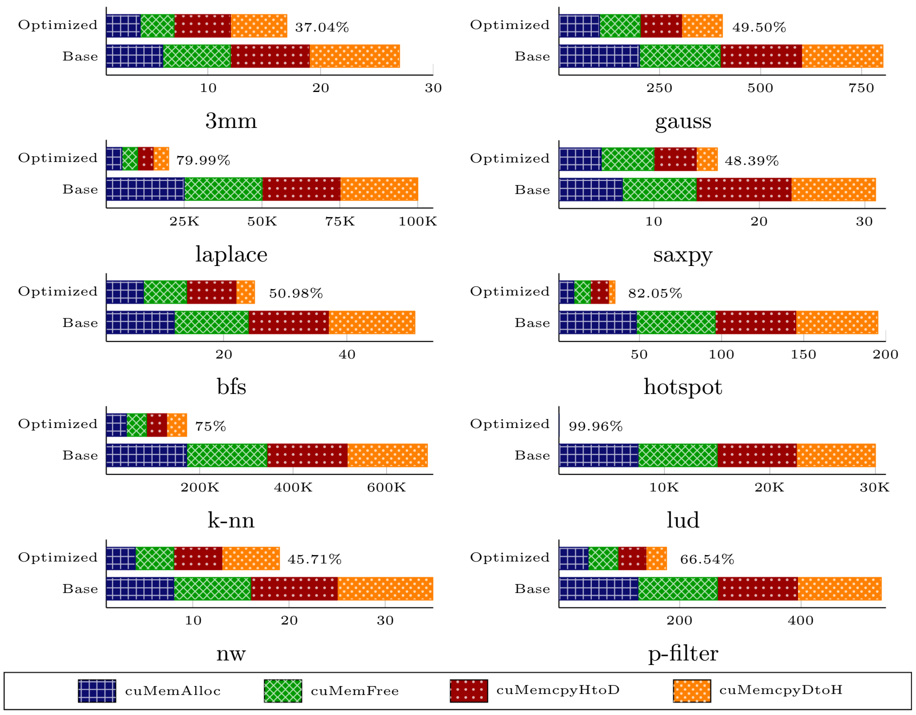


Fig. 3. Number of calls to data transfer CUDA APIs. The % at the tip of optimized code represent reduction in total number of calls when compared to base code.

loop, dramatically reducing the number of times the data was moved between the host and device. In our optimized code, each of the four concerned APIs were only called around 5000 times. To achieve this, our tool added just one *target data* directive. Figure 3 shows the reduction in the number of times the APIs are called, with a maximum of 99.96% reduction in LU-Decomposition and a minimum of 45.71% reduction in Needleman Wunsch.

We next calculated the time it took to perform the data transfer and the time taken to execute the kernel computations. The result is shown in Fig. 4. It can be seen that the kernel execution time in both base and optimized code is almost identical. This is expected, as we did not make any changes to the kernel code itself. In Figs. 4(A), 4(B), 4(E), 4(I) and 4(J), we observe that the majority of the execution time is consumed in the kernel computation rather than in data management. However, the K-Nearest Neighbor algorithm, Fig. 4(G), spends 4.382 s in cuMemAlloc, 2.958 s in cuMemFree, 0.209 s in cuMemcpyHtoD and 0.193 s in cuMemcpyDtoH, with overall 7.743 s for data management, which is almost 91x the kernel computation time of 0.085 s. After applying our optimization, data management was improved by 71% to just 2.26 s. In Fig. 4(G) we do not even see the compute bar as it is insignificant compared to the data management time.

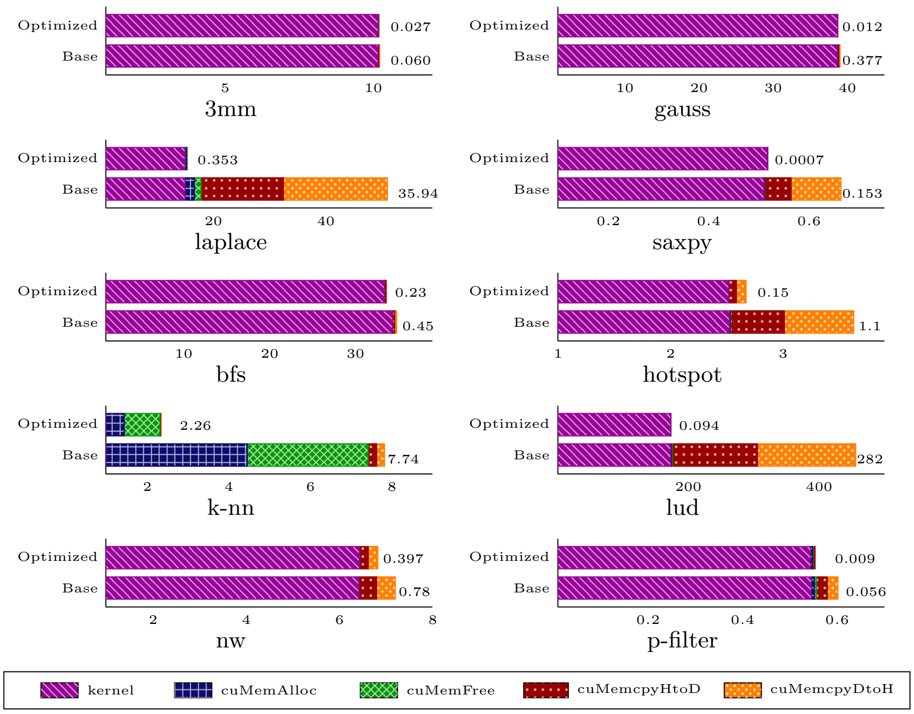


Fig. 4. Time taken (in sec) for different data management APIs and kernel computation time on V100 GPU. The numbers at the tip of each graph represent the time taken for data transfer only (in sec).

For the LU-Decomposition, Fig. 4(H), the base case requires 282.514 s for data management, which is almost 1.63x the kernel compute time of 173.441 s. But after our optimization it consumes only 94.28 ms, which is a 99.96% improvement over the base case. We also observe considerable improvement in data transfer time for Laplace (99.01%), SAXPY (99.54%), HotSpot (85.95%) and Particle Filter (83.96%).

6 Related Work

Optimizing GPU memory management where data movement must be managed explicitly has been explored in a variety of research. Jablin *et al.* [18] provide a fully automatic system for managing and optimizing CPU-GPU communication for CUDA programs. Gelado *et al.* [13] present a programming model for heterogeneous computing to simplify and optimize GPU data management. Recently Barua *et al.* [1] introduce static analysis of explicit data transfers already inserted into an OpenMP code. Current research in the field do not provide insight into utilizing data reusability on GPU for implicitly managed data between multiple kernels.

Although there are several studies on data placement in heterogeneous memory system, like Dullor *et al.* [10] or Yu *et al.* [33], unfortunately they ignore the impact of implicit data transfer in unified memory. Recently Li *et al.* [20] have introduced optimizations to improve the performance of unified memory and their work can be used in extension to our tool. Other related research on fully automatic GPU offloading of code by Mishra *et al.* [23], Mendonça *et al.* [22] and Poesia *et al.* [28], can take benefit from our research to add data reuse optimization in their tool, which would further reduce the barriers to use of GPUs for scientific computing.

7 Conclusion

Careful management of data and its mapping between host and device is critical for the use of accelerators in HPC, given the high cost of data motion. The complexities involved are a deterrent to the exploitation of GPUs. The optimization introduced in this paper may result in a significant reduction in the amount of data transferred between CPU and GPU and hence improve the overall execution time of codes that relied on implicit data transfer. It thus contributes to the ease of use of OpenMP by avoiding the penalty often associated with implicit transfers. To the best of our knowledge, this is the first tool to apply static analysis in order to identify and exploit data reuse between offload kernels in an OpenMP program. The same strategy could clearly easily be used to optimize an OpenMP compiler's handling of such transfers. By producing modified source code, as we have chosen to do, we give the user the option of accepting, improving or rejecting our generated code, and we enable them to use their compiler of choice for further building their code.

8 Future Work

The tool is under constant improvement, and a number of extensions are planned or already under way:

- We are working on exploiting unified memory for data management between the host and the device.
- We also need to further analyze data usage to determine which data can be pinned to the GPU, to reduce multiple data allocations. We are also considering how to handle data between multiple GPUs.
- Discussion is also going on for managing complex structures, sub-structures and sub-arrays and handling data between multiple GPUs as well.
- Most importantly, we are planning to improve our proximity analysis to analyze kernels beyond a single function. This could be facilitated by extending the `target data` directive of OpenMP, so that the user only prescribes the data region without the need to explicitly define the map clause. This would let the compiler automatically handle the data movement, potentially reusing data on the GPU. Any kernel called from such a data region would be considered a candidate for data reuse and be analyzed as such.

Acknowledgement. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the SeaWulf computing system, which was made possible by a $1.4M National Science Foundation grant (#1531492). Special thanks to our colleague Dr. Chunhua Liao from Lawrence Livermore National Laboratory for his initial feedback and helpful discussions.

References

1. Barua, P., Shirako, J., Tsang, W., Paudel, J., Chen, W., Sarkar, V.: OMPSan: static verification of OpenMP's data mapping constructs. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 3–18. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_1
2. Bercea, G.T., et al.: Implementing implicit OpenMP data sharing on GPUs. In: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, pp. 1–12 (2017)
3. C++ Heterogeneous-Compute Interface for Portability (2016). https://github.com/ROCm-Developer-Tools/HIP
4. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 44–54. IEEE (2009)
5. Clang 8.0 (2019). http://releases.llvm.org/8.0.1/tools/clang/docs/index.html
6. Clang, Libtooling (2019). http://clang.llvm.org/docs/LibTooling.html
7. Consortium, O., et al.: OpenMP specification version 5.0 (2018)
8. Cray, C.: C++ reference manual, s-2179 (8.7). Cray Research (2019). https://pubs.cray.com/content/S-2179/8.7/cray-c-and-c++-reference-manual/openmp-overview
9. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
10. Dulloor, S.R., et al.: Data tiering in heterogeneous memory systems. In: Proceedings of the Eleventh European Conference on Computer Systems, pp. 1–16 (2016)
11. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using GPU. In: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 1–6. IEEE (2008)
12. GCC Support for the OpenMP Language (2019). https://gcc.gnu.org/wiki/openmp
13. Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., Hwu, W.M.W.: An asymmetric distributed shared memory model for heterogeneous parallel systems. In: Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 347–358 (2010)
14. Goodrum, M.A., Trotter, M.J., Aksel, A., Acton, S.T., Skadron, K.: Parallelization of particle filter algorithms. In: Varbanescu, A.L., Molnos, A., van Nieuwpoort, R. (eds.) ISCA 2010. LNCS, vol. 6161, pp. 139–149. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24322-6_12
15. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77220-0_21

16. Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., Stan, M.R.: Hotspot: a compact thermal modeling methodology for early-stage VLSI design. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **14**(5), 501–513 (2006)
17. Intel C++ Compiler Code Samples (March 2019). https://software.intel.com/en-us/code-samples/intel-c-compiler
18. 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: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 142–151 (2011)
19. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, p. 75. IEEE Computer Society (2004)
20. Li, L., Chapman, B.: Compiler assisted hybrid implicit and explicit GPU memory management under unified address space. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–16 (2019)
21. LLVM Support for the OpenMP Language (2019). https://openmp.llvm.org
22. Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., Pereira, F.M.Q.: DawnCC: automatic annotation for data parallelism and offloading. ACM Trans. Archit. Code Optim. (TACO) **14**(2), 13 (2017)
23. Mishra, A., Kong, M., Chapman, B.: Kernel fusion/decomposition for automatic GPU-offloading. In: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, pp. 283–284. IEEE Press (2019)
24. Mishra, A., Li, L., Kong, M., Finkel, H., Chapman, B.: Benchmarking and evaluating unified memory for OpenMP GPU offloading. In: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, pp. 1–10 (2017)
25. Nvidia, C.: Nvidia cuda c programming guide. Nvidia Corp. **120**(18), 8 (2011)
26. NVIDIA Tesla: Nvidia tesla v100 GPU architecture (2017)
27. OpenMP Compilers & Tools (April 2019). https://www.openmp.org/resources/openmp-compilers-tools
28. Poesia, G., Guimarães, B., Ferracioli, F., Pereira, F.M.Q.: Static placement of computation on heterogeneous devices. Proc. ACM Program. Lang. **1**(OOPSLA), 50 (2017)
29. Poesia, G., Guimarães, B.C.F., Ferracioli, F., Pereira, F.M.Q.: Static placement of computation on heterogeneous devices. Proc. ACM Program. Lang. **1**(OOPSLA), 50:1–50:28 (2017). Article 50
30. Seawulf, Computational Cluster at Stony Brook University (2019). https://it.stonybrook.edu/help/kb/understanding-seawulf
31. Vazhkudai, S.S., et al.: The design, deployment, and evaluation of the coral pre-exascale systems. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 661–672. IEEE (2018)
32. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC—first experiences with real-world applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32820-6_85
33. Yu, S., Park, S., Baek, W.: Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In: Proceedings of the International Conference on Supercomputing, pp. 1–10 (2017)