CADI: Semi-Automatic Insertion of Host-Device Data Transfer Directives Based on Code Analysis

Koki Watanabe
Tohoku University
Sendai, Miyagi, Japan
watanabe.koki.t2@dc.tohoku.ac.jp

Masatoshi Kawai
Tohoku University
Sendai, Miyagi, Japan
m.kawai@tohoku.ac.jp

Junji Fukuhara
DENSO CORPORATION
Japan
junji.fukuhara.j8j@jp.denso.com

Tomoyoshi Funazaki
DENSO CORPORATION
Japan
tomoyoshi.funazaki.j6j@jp.denso.com

Seiji Nishimura
DENSO CORPORATION
Japan
seiji.nishimura.j4b@jp.denso.com

Sichen Tao
Tohoku University
Sendai, Miyagi, Japan
sichen.tao@tohoku.ac.jp

Hiroyuki Takizawa
Tohoku University
Sendai, Miyagi, Japan
takizawa@tohoku.ac.jp

I. Introduction

Heterogeneous computing systems that combine CPUs with accelerators such as GPUs are now widely used in high-performance computing to meet growing demands for computational performance and energy efficiency. As shown in Fig. 1, the host CPU and accelerator devices typically maintain separate memory spaces. Therefore, data must be transferred between them before executing computations on the device. Since host-device data transfer significantly affects application performance, efficient data movement management is essential.

Existing data transfer methods can be classified into two types: manual management and automatic management. In manual management, programmers explicitly specify data transfer directives, which eliminates unnecessary transfers and achieves high execution performance. However, this requires accurately specifying transfer targets and timing, increasing development effort and the risk of programming errors. Automatic management, such as Unified Memory and Managed Memory, instead eliminates the need for programmers to manage transfers explicitly, reducing programming effort. This convenience comes at the cost of runtime overhead from on-demand data migration, and such methods are often limited to specific hardware environments.

Consequently, it remains difficult to achieve both the high performance of manually optimized data transfer and the convenience of automatic management. To address this trade-off, we adopt a semi-automatic approach that automatically identifies transfer targets and memory regions while allowing programmers to specify transfer directions for performance optimization. Based on this approach, we present CADI (Code Analysis-based Data-transfer directive Insertion), which combines static code analysis with runtime memory tracking.

II. Related Work

Among existing approaches, DawnCC automatically inserts parallelization and data transfer directives through static analysis without programmer intervention. However, because it relies solely on static analysis, it is difficult to handle programs that require runtime information, such as indirect memory accesses (e.g., x[y[i]]).

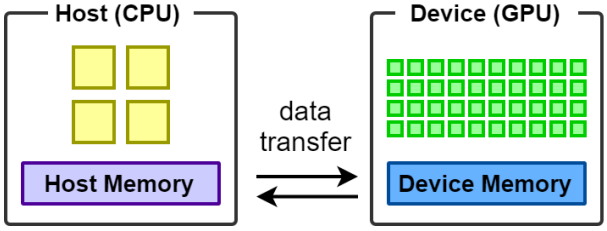


Fig. 1: Host–Device Data Transfer

Unlike DawnCC, CADI combines static code analysis with runtime memory tracking, enabling it to handle cases where static analysis alone is insufficient.

III. Proposed Method

A. Overview of CADI

As an example of manual implementation, Code 1 shows a GPU-parallelized OpenACC program in which lines 1 and 2 specify the data transfer directives. These directives define the data regions and transfer directions required for device execution. CADI automatically inserts these directives that would otherwise be written manually by first obtaining the information required to generate them.

CADI takes a semi-automatic approach by dividing the required information between automatic analysis and programmer's manual guidance. A data transfer directive requires three

Code 1: Example of Data Transfer Directives

```
1 #pragma acc data copyout(p1[0:N]) \
2                  copyin(p2[0:N])
3 {
4   #pragma acc kernels
5   #pragma acc loop independent
6   for(int i=0; i<N; i++)
7     p1[i] = p2[i];
8 }
```

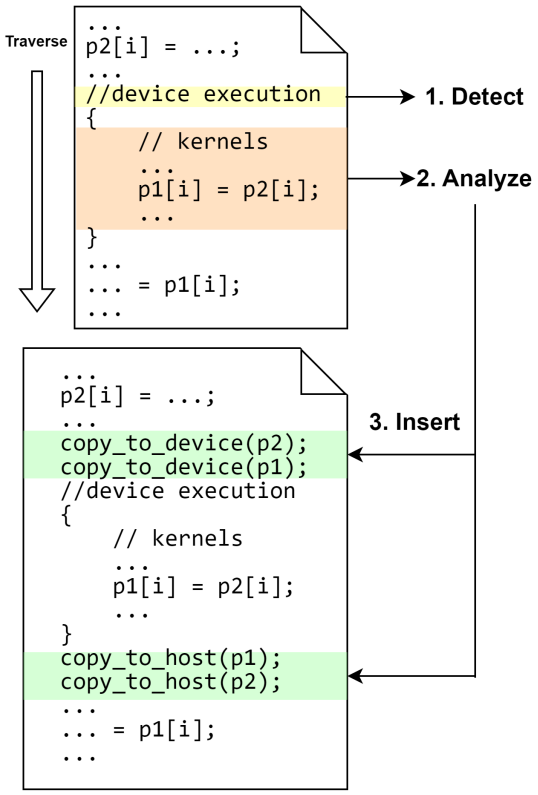


Fig. 2: Code Analysis Procedure

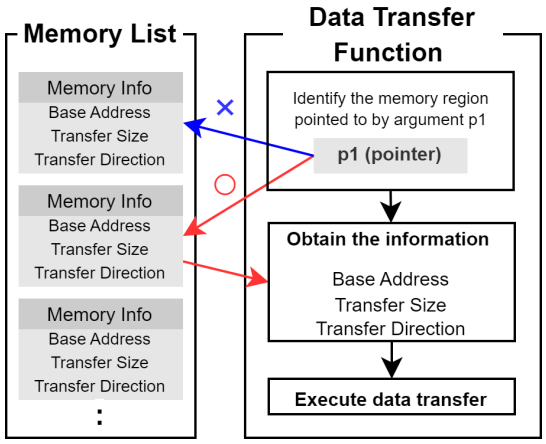


Fig. 3: Runtime Data Transfer Process

pieces of information: the base address of the target memory region, the transfer size, and the transfer direction. CADI obtains the base address and transfer size automatically, while the programmer specifies the transfer direction. To obtain the base address and transfer size, CADI first extracts pointers used in device computation through static code analysis. At runtime, these pointers are used to identify the corresponding memory regions, from which the information is retrieved.

B. Code Analysis

The static analysis procedure is illustrated in Fig. 2. CADI traverses the Abstract Syntax Tree (AST) to detect regions executed on the device and extract the pointers used in device computation. It then automatically inserts the data transfer functions, eliminating the need for programmers to write directives manually.

TABLE I: Evaluation Environment

Item	Specification
CPU	AMD EPYC 7352 (2.3 GHz, 24 cores/socket, 2 sockets)
GPU	NVIDIA A100 80GB
Memory	512GB DDR4
PCI Express	PCI Express Gen4 x16
OS	Rocky Linux 8.9
Compiler	NVIDIA HPC Compiler 25.3
CUDA Toolkit	CUDA Toolkit 12.8
NVIDIA Driver	NVIDIA Driver version 580.126.20

C. Runtime Data Transfer

At runtime, CADI tracks dynamically allocated memory regions to obtain information unavailable through static analysis alone. Fig. 3 illustrates the runtime processing of a data transfer function. When the function is invoked, CADI identifies the memory region corresponding to the pointer extracted during static analysis. It then retrieves the base address, transfer size, and transfer direction, and performs the appropriate data transfer.

IV. Evaluation

We evaluated the execution performance of CADI. Three approaches were compared: Manual, in which all data transfer directives were implemented manually; CADI; and Managed Memory, which provides fully automatic data management. For both Manual and CADI, transfer directions were manually optimized. The evaluation environment is shown in Table I. The dynamic version of the Himeno benchmark was used for evaluation. Performance was measured as the average GFLOPS over five executions.

The results are shown in Fig. 4. Compared with Manual, CADI achieved nearly identical execution performance, with performance degradation of less than 1% for all problem sizes. This result suggests that the overhead introduced by CADI's runtime memory tracking is negligible.

Compared with Managed Memory, CADI achieved higher performance as problem size increased, indicating that CADI is particularly effective for applications involving large-scale data transfers.

V. Conclusion

This research presented CADI, a foundation for semi-automatic insertion of data transfer directives while preserving programmer-guided performance optimization. CADI combines static code analysis with runtime memory tracking to identify transfer targets and memory regions, and uses programmer-provided transfer-direction hints to avoid unnecessary data movement. The current prototype achieved performance comparable to manually optimized implementations, showing that its analysis and runtime mechanisms introduce little overhead. By further automating transfer-direction inference, expanding support for memory allocation patterns and

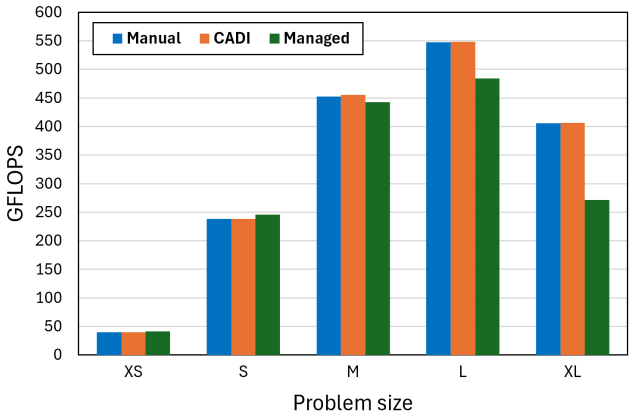


Fig. 4: Performance Comparison Using the Himeno Benchmark

data structures, and extending interprocedural analysis, CADI is expected to substantially reduce transfer annotations while improving performance through more precise data movement.

ACKNOWLEDGMENT

This work was partially supported by collaborative research funding from DENSO CORPORATION. The authors also acknowledge the use of ChatGPT, developed by OpenAI to assist with proofreading and enhance manuscripts clarity. The authors are accountable for contents.