The Journal of Supercomputing (2021) 77:4957–4987
https://doi.org/10.1007/s11227-020-03452-2

Automatic translation of data parallel programs for heterogeneous parallelism through OpenMP offloading

Farui Wang[1] · Weizhe Zhang[1] · Haonan Guo[1] · Meng Hao[1] · Gangzhao Lu[1] · Zheng Wang[2]

Accepted: 11 October 2020 / Published online: 29 October 2020

Abstract
Heterogeneous multicores like GPGPUs are now commonplace in modern computing systems. Although heterogeneous multicores offer the potential for high performance, programmers are struggling to program such systems. This paper presents OAO, a compiler-based approach to automatically translate shared-memory OpenMP data-parallel programs to run on heterogeneous multicores through OpenMP offloading directives. Given the large user base of shared memory OpenMP programs, our approach allows programmers to continue using a single-source-based programming language that they are familiar with while benefiting from the heterogeneous performance. OAO introduces a novel runtime optimization scheme to automatically eliminate unnecessary host–device communication to minimize the communication overhead between the host and the accelerator device. We evaluate OAO by applying it to 23 benchmarks from the PolyBench and Rodinia suites on two distinct GPU platforms. Experimental results show that OAO achieves up to 32× speedup over the original OpenMP version, and can reduce the host–device communication overhead by up to 99% over the hand-translated version.

Keywords Heterogeneous computing · Source-to-source translation · OpenMP offloading · Compilation optimization · GPUs

✉ Weizhe Zhang
wzzhang@hit.edu.cn

Farui Wang
wangfarui@hit.edu.cn

Zheng Wang
z.wang5@leeds.ac.uk

[1] School of Computer Science and Technology, Harbin Institute of Technology, Harbin, HL, China

[2] School of Computing, University of Leeds, Leeds, UK

1 Introduction

Heterogeneous multicores, as represented by the GPUs, are now pervasive in computing systems because of their energy-efficient high performance. Such a potential can only be unlocked if the running software has been suitably parallelized to match the underlying hardware. Unfortunately, developers struggle to program heterogeneous multicores due to the complexity in offloading computation and communication management between the host and the accelerator device.

Numerous programming models have been proposed to address the programming issue of heterogeneous systems, including Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL), Open Accelerators (OpenACC), and more recently—Open Multi-Processing Offloading (OpenMP Offloading) [21–23]. These approaches enable newly developed codes to run on heterogeneous devices. However, they offer little help in addressing the problem of porting legacy programs to heterogeneous devices because programmers still need to painstakingly modify the existing code to use a heterogeneous programming model.

Compiler-based source-to-source translators offer a viable solution and roadmap for porting legacy parallel code to run on heterogeneous computing devices. Some existing work targets CUDA code generation [2, 11, 18, 30]. However, this kind of existing work has a serious performance portability issue as an application implemented in CUDA, by definition, is not portable to non-NVIDIA systems. Other existing work [19, 26, 33, 34] generates OpenCL code that can run on a wide range of parallel hardware including GPUs, CPUs, and FPGAs. Given that OpenCL remains as a low-level programming language that exposes many hardware details, maintaining the generated code is often too difficult for non-expert programmers.

The emerging OpenMP Offloading standard [21–23] offers a promising approach to port legacy OpenMP programs to heterogeneous devices using simple language pragmas with little modification to existing code while preserving the advantage of low maintenance costs given by the simplicity of OpenMP [20]. Compared with CUDA or OpenCL, this standard allows programmers to work on a language that they are familiar with using a few intuitive pragmas to annotate their code. Compared with OpenACC, this standard is supported by more commonly used compilers. Thus, OpenMP Offloading provides existing OpenMP programs with a simple upgrade path to heterogeneous parallelism using pragmas. Although promising, OpenMP Offloading still requires manual optimization of the data transmission to achieve good performance.

The DawnCC compiler [15, 17] is among the first attempts to leverage OpenMP Offloading for heterogeneous computing. This compiler translates sequential C into OpenACC or OpenMP Offloading. However, DawnCC does not address the communication optimization problem between the host CPU and the heterogeneous accelerator well, because of the lack of inter-procedural data transmission optimization, which is often responsible for the performance bottleneck. Moreover, DawnCC often does not choose the right offloading directives, leading to suboptimal performance. As a result, the code generated by DawnCC often delivers worse performance than the original OpenMP running on a shared-memory parallel machine. This drawback discourages the adoption of the technique on a broader scale.

This work aims to provide a better approach for leveraging OpenMP Offloading for heterogeneous computing. We present OpenMP Automatic Offloading (OAO), a source-to-source framework that automatically translates OpenMP parallel loops to use OpenMP Offloading pragmas. Instead of performing simple code translation, we go further by developing a runtime system to optimize the data communication between the host CPU and the accelerator automatically. By precisely modeling the consistency state and its transition of a data buffer, our runtime eliminates redundant data transmissions, on-the-fly, for not just simple loops but also complex data structures and nested function calls. We show that OAO is highly effective in generating efficient OpenMP Offloading code to run on heterogeneous GPUs. We demonstrate the benefit of OAO by it to 23 OpenMP benchmarks from the PolyBench and Rodinia suites. We compare OAO with DawnCC and manually-translated codes on two distinct GPU platforms with a K40 or a 2080Ti GPU. Experimental results show that OAO achieves up to 32× speedup over the original OpenMP version. Moreover, it can reduce the host–device communication time by up to 99% compared with the manually-translated version. We show that OAO can also handle benchmarks that DawnCC fails on, with significantly better performance improvement.

This paper makes the following technical contributions:

- We propose the first source-to-source tool that directly translates legacy OpenMP programs into OpenMP Offloading programs without manual intervention.
- We present a novel algorithm to optimize the host–device communication by levering the consistency states of the program. Unlike prior work, our approach can work on complex data structures and nested function calls.

The OAO source-to-source translator framework is publicly available at https://github.com/ruixueqingyang/OAO-Translator.

The remainder of this work is organized as follows: Sect. 2 introduces the motivation and overview of the OAO system. Section 3 describes the OAO runtime library (OAORT) and the minimum transmission algorithm. Section 4 proposes the OAO translator with algorithms to insert OAORT APIs. Section 5 describes the experimental setup. Section 6 presents and analyzes the experimental results. Section 7 provides the related work. Finally, Sect. 8 concludes the paper and discusses future work.

2 Background and overview

2.1 OpenMP offloading

Since version 4.0, OpenMP standard introduced new offloading constructs for heterogeneous computing. These offloading constructs allow the program to specify which regions of code and data to be mapped to run on an accelerator.

Figure 1b gives a simple use case of OpenMP Offloading constructs. Here, the user program starts execution on a host (e.g., CPU) device, where offloading to an accelerator is performed when entering a *target* region specified by the `target` pragma. A target region maps variables allocated on the host memory to the device memory, e.g., the GPU global memory. The implementation of target regions

```
#pragma omp parallel for
for(int i = 0; i < N; i++){
    v3[i] += v2[i] + v1[i];
}
#pragma omp parallel for
for(int i = 0; i < N; i++){
    v5[i] += v4[i] + v3[i];
}
```

(a) OpenMP CPU code snippet

```
#pragma omp target data map(tofrom: v1[:N], v2[:N], v3[:N])
{
    #pragma omp target teams distribute parallel for
    for(int i = 0; i < N; i++){
        v3[i] += v2[i] + v1[i];
    }
}
#pragma omp target data map(tofrom: v3[:N], v4[:N], v5[:N])
{
    #pragma omp target teams distribute parallel for
    for(int i = 0; i < N; i++){
        v5[i] += v4[i] + v3[i];
    }
}
```

(b) OpenMP Offloading code snippet

```
#pragma omp target enter data map(to: v1[:N], v2[:N], v3[:N])
#pragma omp target teams distribute parallel for
for(int i = 0; i < N; i++){
    v3[i] += v2[i] + v1[i];
}
#pragma omp target enter data map(to: v4[:N], v5[:N])
#pragma omp target teams distribute parallel for
for(int i = 0; i < N; i++){
    v5[i] += v4[i] + v3[i];
}
#pragma omp target exit data map(from: v3[:N], v5[:N]) map(delete:
     v1[:N], v2[:N], v4[:N])
```

(c) Optimal OpenMP Offloading code snippet

Fig. 1 Example for OpenMP CPU to OpenMP Offloading translation and optimization

may include transmitting data between host and device and launching a GPU kernel to execute the target code region. The `teams` directive can be used to spawn a league of teams, each consists of multiple OpenMP threads. It is to note that any two threads from different teams cannot communicate in any native way, e.g., no barrier can be placed between threads from different teams. This feature ensures the accelerator implementation can map individual teams to run on independent execution units.

The `distribute` pragma can be used to partition the loop iterations into chunks to be allocated to teams. Note that function calls and global variable references are allowed in target regions, but they increase the difficulties for performance host–device communication optimization. The `target data map` directive specifies the variable mapping and unmapping operations at the beginning and end of the brace region following the directive. The `target enter data map` and `target exit data map` directives define the variable mapping and unmapping operations at the beginning and end of the region between these two directives, respectively. The `to` pragma indicates data transmission from the host to the device when the variables are mapped to the device memory. The `from` pragma means copying data from the device to the host when the variables are unmapped form the device memory. The `tofrom` pragma integrates the functions of `to` and `from` pragmas. The `delete` pragma means unmapping variables form the device memory without data copy.

2.2 Motivation

As a motivation example, consider translating the example code given in Fig. 1a. Translating the two OpenMP data-parallel loops to use OpenMP Offloading for heterogeneous computing is straightforward. A naive solution and an optimal solution are given in Fig. 1b, c, respectively.

Compared to the naive solution, inserting the appropriate data transmission directives to achieve good performance is non-trivial. For example, the translation given in Fig. 1b contains redundant transmissions, such as all "`from`" transmissions at line 1, "`to`" transmission for v3 at line 8 and "`from`" transmission for v4 at line 8. These redundant host–device communications results in 1.6× slowdown compared to the version given in Fig. 1c. Developers can eliminate these redundant data transmissions only if they understand the following knowledge: (1) read and write operations of variables within and between these two loops, (2) the consistency states of variables before and after these two loops, and (3) the consistency states of variables required by these two loops. This is complex and tedious for developers. OAO is designed to remove redundant host–device communications by executing the right data transmission directives at the right place automatically.

2.3 Overview of our approach

As depicted by Fig. 2, OAO consists of two components: a source-to-source code translator and a runtime library. The code translator translates OpenMP (or OMP in

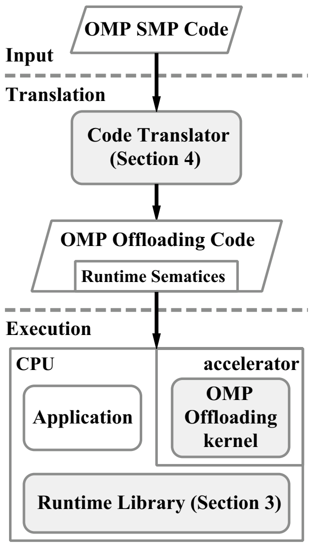

Fig. 2 Overview of OAO

short) symmetric multiprocessing (SMP) constructs to OMP Offloading code when it is possible to do so. The translated code is compiled and linked with the runtime library. During execution, the runtime automatically determines and executes essential data transmissions without redundant data transmissions through the minimum transmission algorithm (Algorithm 2).

As shown in Table 1, compared with existing source-to-source translators, only our OAO can translate OpenMP code to OpenMP Offloading code, whose intra-procedural and inter-procedural data transmission are both optimized. Another difference is that only our work and Grewe et al. [19] use both the runtime and static code translator, while other existing work only uses the static code translator. Grewe et al. use the runtime to determine where to run the program (CPU or GPU). We use our runtime to optimize data transmissions.

A key innovation of our approach is using the consistency state, state transition function, and consistency state constraint to perform host–device communication optimization in our runtime. We model the data transmission directive and the variable reference, which may change the consistency state, as the state transition function. Our key insight is that the consistency of shared data in the host and device memory is guaranteed as long as the consistency state satisfies the consistency state constraint. We propose a novel algorithm to derive the essential state transition function, which transits the consistency state to solve the consistency state constraint. With the algorithm in place, our runtime can execute the right data transmission directive corresponding to the essential state transition function and avoid redundant data transmissions. When the variable is referenced, our runtime uses the corresponding state transition function to update the consistency state maintained in runtime. We implement the algorithm and update operation as data transmission and consistency state update semantics, which should be inserted to OMP Offloading programs properly. Our runtime library is detailed in Sect. 3.

Table 1 Comparison with existing source-to-source translators for accelerators

Translator	Input	Output	Method	Data transmission optimization type
C-to-CUDA [2]	C	CUDA	Static code translator	None
PPCG [30]	C	CUDA	Static code translator	Intra-procedural
BONES [18]	C	CUDA OpenCL	Static code translator	Intra-procedural
OpenMPC [11]	OpenMP	CUDA	Static code translator	None
Grewe et al. [19]	OpenMP	OpenCL	Static code translator runtime	None
DawnCC [15, 17]	C	OpenMP Offloading, OpenACC	Static code translator	Intra-procedural
OAO (our work)	OpenMP	OpenMP Offloading	Static code translator runtime	Intra-procedural inter-procedural

To generate code to use OpenMP Offloading constructs, we first construct the extended control-flow graph for each function in OMP SMP programs. From the control-flow graph, we collect and analyze variable reference information through compile-time static analysis techniques. The analysis is used to insert the right runtime API functions in the right places. We describe our code translator in Sect. 4.

3 Runtime library

Algorithm 1 explains the workflow of the OAO Runtime Library (OAORT) that aims to minimize host-accelerator data transmission while guaranteeing data consistency. The OAORT initializes and maintains the consistency states of variables (line 1). Then, the OAORT determines and executes the minimum data transmission operations according to the consistency state constraints required by the following code snippet (line 2). After the code snippet, the OAORT updates the consistency states changed by the code snippet (line 4). As a final step, the OAORT deletes the maintained consistency states (line 5). The initialization and deletion are introduced in Sect. 3.1. The data transmission and consistency state update are introduced in Sects. 3.2 and 3.3 separately.

Algorithm 1: Runtime workflow

```
Input: OpenMP Offloading code with OAO Runtime API inserted
1 Initialize and maintain consistency states of variables with API functions such as
  OAOSaveArrayInfo(), OAOMalloc(), and OAONewInfo()
2 Determine and execute the minimum data transmission operations for variables
  with API function OAODataTrans()
3 Run the code snippet that references these variables
4 Update consistency states of variables with API functions OAOStTrans()
5 Delete consistency states maintained in OAO Runtime with API functions such as
  OAODeleteArrayInfo(), OAOFree(), and OAODeleteInfo()
```

3.1 Tracking consistency states

Guaranteeing data consistency is the fundamental objective of the OAORT. We define the *State* to represent the consistency states of variables. The consistency *State* is the foundation of the OAORT and minimum transmission algorithm.

3.1.1 Consistency states

The CPU and the accelerator have two independent memory spaces. Variables may reside in either or both of these two memory spaces. So, for a variable, there are three situations: residing in CPU memory (HOST_ONLY), residing in accelerator memory (DEVICE_ONLY), and residing in both memory. DEVICE_ONLY is ignored because these accelerator local variables do not need to be transmitted between the CPU and accelerator. For the third situation, there are

Table 2 All possible consistency states of a variable

State	Bit2: accelerator copy valid	Bit1: CPU copy valid	Bit0: has accelerator copy
HOST_ONLY	0	1	0
HOST_NEW	0	1	1
DEVICE_NEW	1	0	1
SYNC	1	1	1

three cases: The CPU copy is valid (HOST_NEW), the accelerator copy is valid (DEVICE_NEW), and both copies are valid (SYNC). We define the consistency *State* to abstract these four cases, as shown in Table 2 (Definition 1).

Definition 1 ***State*** is a 3-bit binary number. Bit0 suggests whether the allocation unit has an accelerator copy (Bit0 = 1) or not (Bit0 = 0). Bit1 indicates whether the CPU copy is valid (Bit1 = 1) or invalid (Bit1 = 0). Bit2 indicates whether the accelerator copy is valid (Bit2 = 1) or invalid (Bit2 = 0).

When the current *State* and the later program's requirement of *State* are known, we can derive the minimum data transmission directive without redundant transmissions through Algorithm 2, which is designed in Sect. 3.2.3. This is our core insight.

3.1.2 Using allocation units as granularity

The OAORT tracks consistency at the granularity of allocation units [8] and maintains a consistency *State* for each allocation unit. In C and C++, an allocation unit is a contiguous region of memory allocated as a single unit. Memory blocks returned from `malloc()`, local variables, and global variables are all examples of allocation units [8]. Our work solely focuses on data parallel programs. For this kind of program, usually, all elements of an allocation unit are accessed if the allocation unit is referenced by a parallel region. Thus, using the allocation units as granularity transmits little redundant data and introduces little overhead. In turn, this helps to handle pointer aliasing and to prevent complex fine-grained symbolic range analysis adopted by DawnCC [15].

To maintain the information of allocation units, we define *MemBlock* and *MemEnv* in Definition 2 and 3, respectively.

Definition 2 ***MemBlk*** refers to a set of characteristics representing an allocation unit (Eq. 1). ***Begin*** is the starting memory address. ***Length*** is the length of the allocation unit. ***ElemSize*** is the element size. ***State*** has been defined above.

$$MemBlk = \{Begin, Length, ElemSize, State\} \tag{1}$$

Definition 3 ***MemEnv*** refers to the set of all *MemBlk*s (Eq. 2).

$$MemEnv = \{MemBlk_1, \ldots, MemBlk_p\} \tag{2}$$

When a pointer (*ptr*) accesses an allocation unit, OAORT searches in the *MemEnv* and find the accessed allocation unit *MemBlk*, which satisfies Eq. 3.

$$Begin \leq ptr \leq Begin + Length - 1 \tag{3}$$

The OAORT provides several API functions to track the *State* of allocation units (Table 3). We insert `OAOSaveArrayInfo` function in the source code to track global and stack memory. The `OAOMalloc` and `OAONewInfo` functions replace `malloc()` and `new`, respectively, to allocate and track heap memory. These three functions build new *MemBlk*s in the *MemEnv*. We insert `OAODeleteArrayInfo` at the end of the variable scope. The `OAOFree` and `OAODeleteInfo` functions replace `free()` and `delete`, respectively. These three functions remove corresponding *MemBlk*s from the *MemEnv*.

For NVIDIA GPU, we optimize the memory allocation specifically to fully exploit bandwidth between CPU and GPU. Pageable memory shows high bandwidth when memory size is relatively small, whereas pinned memory shows high bandwidth when memory size is large. According to experimental results, the threshold is set to 128 KB. When memory block is not larger than 128 KB, `malloc()` is used inside `OAOMalloc` to allocate pageable memory. Otherwise `cudaMallocHost()` is used to allocate pinned memory. In `OAOFree`, `free()` and `cudaFreeHost()` functions are used to release corresponding memory.

3.2 Data transmission semantics

We define the consistency state constraint (*Constr*) to represent the requirement of *State*. The data transmission semantic (`OAODataTrans` function) automatically

Table 3 OAO Runtime API functions

Function	Description
`void OAOSaveArrayInfo(void* ptr, size_t length, size_t ElementSize)`	Saving the static array information
`void OAODeleteArrayInfo(void* ptr)`	Removing the static array information
`void* OAOMalloc(size_t length)`	Saving the dynamic array information
`void OAOFree(void* ptr)`	Removing the dynamic array information
`void* OAONewInfo(void* ptr, size_t ElementSize, size_t ElementNum)`	Saving the dynamic array information
`void OAODeleteInfo(void *ptr)`	Removing the dynamic array information
`void OAODataTrans(void* ptr, STATE_CONSTR Constr)`	Determining and performing the minimum data transmission
`void OAOStTrans(void *ptr, STATE_CONSTR StTrans)`	Transiting the consistency state

Table 4 Description of *ConVld* and *ConInVld*

Bit	Requirement
ConVld Bit0	Require the variable to have been mapped to accelerator memory, namely *State* Bit0 = 1, (*ConVld* Bit0 = 1) or not require (*ConVld* Bit0 = 0)
ConVld Bit1	Require the CPU copy to be valid, namely *State* Bit1 = 1, (*ConVld* Bit1 = 1) or not require (*ConVld* Bit1 = 0)
ConVld Bit2	Require the accelerator copy to be valid, namely *State* Bit2 = 1, (*ConVld* Bit2 = 1) or not require (*ConVld* Bit2 = 0)
ConInVld Bit0	Require the variable to have been unmapped from accelerator memory, namely *State* Bit0 = 0, (*ConInVld* Bit0 = 0) or not require (*ConInVld* Bit0 = 1)
ConInVld Bit1	Require the CPU copy to be invalid, namely *State* Bit1 = 0, (*ConInVld* Bit1 = 0) or not require (*ConInVld* Bit1 = 1)
ConInVld Bit2	Require the accelerator copy to be invalid, namely *State* Bit2 = 0, (*ConInVld* Bit2 = 0) or not require (*ConInVld* Bit2 = 1)

derives and executes the essential and minimum data transmission directive to guarantee consistency, according to the current *State* maintained in runtime and the *Constr*.

3.2.1 Consistency state constraints

We represent the requirement of *State* with the *Constr* (Definition 4). Table 4 explains each bit of *Constr*. If any bit of *State* is required to be set to 1, we set the corresponding bit of *ConVld* to 1. If any bit of *State* is required to be set to 0, we set the corresponding bit of *ConInVld* to 0.

Definition 4 ***Constr*** refers to a pair of 3-bit binary numbers (Eq. 4). Table 4 explains the different requirements represented by various bit values of *ConVld* and *ConInVld*.

$$Constr = \{ConInVld, ConVld\} \tag{4}$$

To guarantee consistency, different *Constr*s should be satisfied before different READ, WRITE, and memory-free operations. We list specific *Constr*s for these operations in Table 5.

Table 5 Operations and required *Constr*s

Operation	*Constr*
Variable unused	$ConNo = \{111, 000\}$
CPU READ	$ConSEQR = \{111, 010\}$
Accelerator READ	$ConOMPR = \{111, 101\}$
CPU WRITE	$ConSEQW = \{111, 000\}$
Accelerator WRITE	$ConOMPW = \{111, 001\}$
CPU free	$ConFREE = \{010, 010\}$

3.2.2 Consistency states transition functions

We define the consistency state transition function (*TransFunc*) to formalize the *State* transitions caused by different data transmission directives and READ/WRITE operations (Definition 5). The formalization helps derive the essential and minimum data transmission directive.

Definition 5 The form of ***TransFunc*** is defined by Eq. 5. *TransFunc* transits *inState* to *outState*. ***InVld*** and ***Vld*** are a pair of 3-bit binary numbers. The operators in Eq. 5 are Boolean multiplication and Boolean addition. The form of ***TransFunc*** is abbreviated as Eq. 6.

$$\begin{aligned} outState &= TransFunc(inState) \\ &= inState \cdot InVld + Vld \end{aligned} \tag{5}$$

$$TransFunc = \{InVld, Vld\} \tag{6}$$

If any bit of the *outState* requires to be set to 0, then the corresponding bit of *InVld* is set to 0. If any bit of the *outState* requires to be set to 1, then the corresponding bit of *Vld* is set to 1. For the bits without requirements, the corresponding bits of *InVld* and *Vld* are set to 1 and 0, respectively. Thus, the form of *TransFunc* can transit any 3-bit binary number to any 3-bit binary number and express any transition between *State*s.

Each kind of data transmission directive or operation corresponds to a *TransFunc* (Table 6). The transition relationships between *State*s are shown in Fig. 3. If we can derive the essential and minimum *TransFunc*, we can know the essential and minimum data transmission directive, which should be executed to guarantee consistency. We design the derivation algorithm in Sect. 3.2.3.

3.2.3 Minimum transmission algorithm

Based on *State*s, *Constr*s, and *TransFunc*s, we design the algorithm to derive the essential and minimum *TransFunc*. The minimum *TransFunc* only change the bits, which

Table 6 State transition operations and corresponding *TransFunc*s

Operation	*TransFunc*
no data transmission required	$TrNo = \{111, 000\}$
`#pragma omp target enter data map(alloc: ...)`	$TrAlloc = \{111, 001\}$
`#pragma omp target enter data map(to: ...)`	$TrEnTo = \{111, 101\}$
`#pragma omp target update to(...)`	$TrUpTo = \{111, 100\}$
`#pragma omp target update from(...)`	$TrFrom = \{111, 010\}$
`#pragma omp target exit data map(delete: ...)`	$TrDelete = \{010, 010\}$
CPU WRITE	$TrSEQW = \{011, 010\}$
Accelerator WRITE	$TrOMPW = \{101, 100\}$
CPU or accelerator READ	$TrRead = \{111, 000\}$

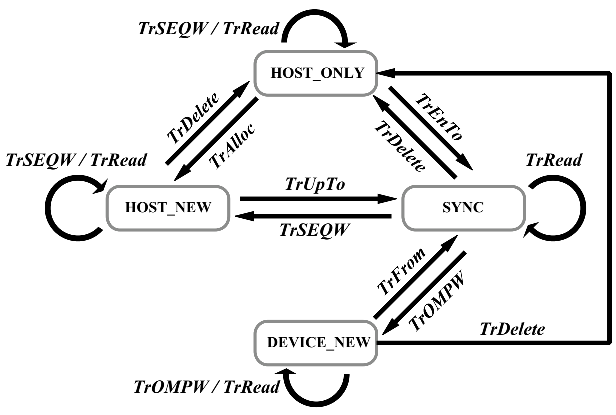

Fig. 3 Consistency state transition relationships

must be changed, to satisfy the *Constr*. This feature prevents redundant data transmissions. In more detail, we define the *MinTrFunc* to represent minimum *TransFunc* (Definition 6).

Definition 6 For the *State* and *Constr*, the **MinTrFunc** refers to the TransFunc that meets the following features. The *Vld* of the *MinTrFunc* only sets the bits, which are 0 in *State* but 1 in *ConVld*, to 1. The *InVld* of the *MinTrFunc* only sets the bits, which are 1 in *State* but 0 in *ConInVld*, to 0.

For example, we assume the *State* = *HOST_NEW* = $\{011\}$ and the *Constr* = *ConOMPR* = $\{111, 101\}$ and derive the *MinTrFunc*. The *ConInVld* = 111 indicates none bits of *InVld* in *MinTrFunc* should be set to 0. Thus, we get *InVld* = 111. The Bit0 and Bit2 of *ConVld* are 1 and we compare these two bits with Bit0 and Bit2 of *State*. The Bit2 of *State* is not equal to the Bit2 of *ConVld*. Thus, we only set Bit2 of *Vld* to 1 and get *InVld* = 100. In summary, we get *MinTrFunc* = $\{111, 100\}$. The *MinTrFunc* = $\{111, 100\}$ = *TrUpTo* corresponds to `#pragma omp target update to` directive, according to Table 6. We get the *MinTrFunc* and the minimum data transmission directive.

```
Algorithm 2: Minimum transmission algorithm
Input: $State$, $Constr = \{ConInVld, ConVld\}$
Output: minimum transition function $MinTrFunc$, minimum data transmission
        directive
1
2 $InVld = \overline{\overline{(State \oplus ConInVld)} \cdot \overline{ConInVld}}$
3 $Vld = (State \oplus ConVld) \cdot ConVld$
4 $MinTrFunc(State) = State \cdot InVld + Vld$
5 Find the minimum data transmission directive corresponding to $MinTrFunc$ in
    Table 6 (the first six lines)
```

According to Definition 6, we propose Algorithm 2 to derive the *MinTrFunc* and minimum data transmission directive. In Algorithm 2, the '$\oplus$' operation takes out bits, which vary in *State* and *ConVld*. The '$\cdot$' operation takes out bits, which are 1 in *ConVld*. Thus, the *Vld* satisfies Definition 6. Similarly, *InVld* satisfies Definition 6. In summary, *MinTrFunc* satisfies Definition 6. The minimum data transmission directive can be determined by looking up Table 6 when *MinTrFunc* is derived.

When the data transmission semantic is called, the minimum data transmission directive is determined through Algorithm 2 and executed automatically. Namely, the essential data transmission, corresponding to the minimum data transmission directive, is determined and executed automatically. With the implementation of Algorithm 2, the OAO Runtime can only execute the essential data transmissions and eliminate redundant transmissions. This is summarized as Question 3 in the experimental part and will be verified by experiments.

3.3 Consistency state update semantics

The WRITE operations in code fragments may change consistency *State*s. Thus, consistency state update semantics (`OAOStTrans` functions) are used to update *State*s maintained in *MemEnv*. The READ operations are not considered because they do not change *State*s. According to the type of WRITE operations, different *TransFunc*s are used to update *State*s (Table 6 [lines 6 and 7]).

4 Code translator

The OAO Translator models each function as an extended control flow graph (CFG) called *SPGraph*. With the *SPGraph*, we analyze the *Constr*s and R/W operations and insert data transmission semantics and consistency state update semantics. We also translates parallel primitives. Using Algorithm 3 5 and Table 7 proposed below, the OAO Translator can translate OMP SMP code into OMP Offloading code. This is summarized as Question 1 in the experimental part and will be verified by experiments.

4.1 *SPGraph* for code translation

To offload data parallel code regions to the accelerator, these code regions should be marked. To handle data transmission and guarantee data consistency between the CPU and the accelerator, the information of variable references on the CPU and the accelerator should be saved separately. For these motivations, we extend the CFG to encode the information required for code translation. We split each data parallel code region as a new node in the CFG. Then, we mark all nodes in the CFG as two types: parallel nodes for nodes of data parallel code regions, and sequential nodes for other nodes. As a final step, we attach variable reference information to the corresponding nodes.

The *SPGraph*, which extends from the CFG, is formally defined by Definitions 7–9. It is important to note that developers should ensure that there is no dependency among the parallel region defined in Definition 8. Each function in the source code is modeled as a *SPGraph*. All the information needed to establish the *SPGraph* can be collected through compile-time static analysis techniques. We choose the sequential and parallel regions as basic units of analysis for two reasons. First, as long as appropriate data transmissions are inserted before a sequential and parallel region, the consistency within the region can be guaranteed. Second, the update of *State*s in *MemEnv* can be delayed until just after the current sequential or parallel region, because the updated *State*s is useful to successive regions rather than the current region.

Definition 7 A **sequential region** is a code fragment that is executed sequentially without branch and outside **#pragma omp parallel** scopes. A sequential region corresponds to a sequential node, denoted by ***SEQ***, in *SPGraph*.

Definition 8 A **parallel region** is a code fragment within a `#pragma omp parallel` scope. A parallel region corresponds to a parallel node, denoted by ***OMP***, in *SPGraph*.

Definition 9 A ***SPGraph*** is a special control flow graph of a function (Eq. 7 and Fig. 4). ***NodeGrp***, the set of all nodes, consists of *SEQGrp* and *OMPGrp*. ***SEQGrp*** is the set of all *SEQ*s. ***OMPGrp*** is the set of all *OMP*s. ***CtlEdge*** is the set of all edges among different nodes.

$$\begin{aligned} SPGraph &= (NodeGrp, CtlEdge) \\ NodeGrp &= SEQGrp \cup OMPGrp \\ CtlEdge &= \{\langle x,y\rangle | x,y \in NodeGrp\} \\ SEQGrp &= \left\{SEQ_1, \dots, SEQ_n\right\} \\ OMPGrp &= \left\{OMP_1, \dots, OMP_m\right\} \end{aligned} \tag{7}$$

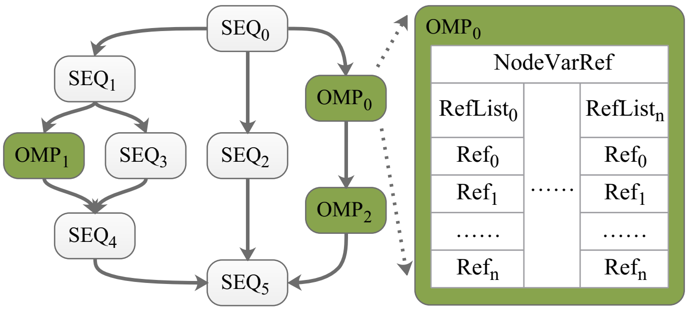

Fig. 4 SPGraph

The variable reference information (*NodeVarRef*), which is collected within a sequential or parallel region, is attached to the corresponding *SEQ* or *OMP* (Definitions 10–12 and Fig. 4).

Definition 10 ***NodeVarRef*** refers to the set of variable reference sequences (*RefList*) in a node.

$$NodeVarRef = \left\{RefList_1, \dots, RefList_n\right\}$$

Definition 11 ***RefList*** refers to a sequence of variable references (*Ref*) of a variable in a node.

$$RefList = \left\{Ref_1, \dots, Ref_m\right\}$$

Definition 12 ***Ref*** is the type of a reference to a variable. *R* represents the READ operation. *W* represents the WRITE operation.

$$Ref = \begin{cases} R; & READ\ operation \\ W; & WRITE\ operation \end{cases}$$

Algorithm 3: Insert OAORT API functions

```
  Input: the $SPGraph$ of a function in source code
  Output: the source code of the funtion with OAORT API funcitons
1 Split each function call in $SEQ$s as a new independent $SEQ$
2 Save information of all function calls in $OMP$s to $NodeVarRef$s
3 foreach variable referenced in all SEQs do
4     foreach $Node \in NodeGrp$ do
5         Call Algorithm 4 to insert data transmission API functions
6
7         Call Algorithm 5 to insert consistency state update API functions
8     end
9 end
```

Based on the *SPGraph*, we propose Algorithm 3 to insert data transmission and consistency state update semantics. First, we preprocess the function calls. Each function call in *SEQ*s is split as a new independent special *SEQ* node (line 1). Each function call in *OMP*s is treated as references to the arguments of the function call (line 2). If an argument is not the pointer type or reference type, then *R* is inserted to the proper position of the corresponding *RefList*. For each argument of pointer type or reference type, if any WRITE operation to the corresponding parameter exists in the called function, *RW* is inserted to the proper position of the corresponding *RefList*, otherwise, *R* is inserted. When function calls are treated as *SEQ*s or variable references, the intra-procedural and inter-procedural data transmission optimizations can be done by inserting data transmission and consistency state update semantics through Algorithms 4 and 5 in Sects. 4.2 and 4.3, respectively. On the contrary, DawnCC does not consider function calls, so it cannot optimize inter-procedural data transmissions.

We apply Algorithm 3 to nearly all functions except *OMP*-called functions, which are called by any *OMP* at least once. The *OMP*-called functions may run on accelerators, whereas OAORT API functions can only run on CPUs. Thus, OAORT API functions cannot be inserted into *OMP*-called functions. The variable consistency of *OMP*-called functions will be guaranteed by OAORT semantics inserted before and after *OMP*-called function calls.

Algorithm 4: Insert data transmission semantics

```
   Input: $Var$, $Node$
   Output: the source code with data transmission semantics
1  Perform static analysis to get the pointer $ptr$ of $Var$
2  if $Node \in OMPGrp$ then
3      Insert API function OAODataTrans(ptr, ConOMPR) before the $Node$
4  else if $Node \in SEQGrp \wedge$ Node is a function call then
5      if the callee function is an OMP-called function then
6          Insert API function OAODataTrans(ptr, ConSEQR) before the $Node$
7      else
8          if the corresponding argument is not pointer type or reference type then
9              Insert API function OAODataTrans(ptr, ConSEQR) the before $Node$
10         end
11     end
12 else
13     Insert API function OAODataTrans(ptr, ConSEQR) before the $Node$
14 end
```

4.2 Handling data transmissions

We proposed Algorithm 4 to determine *Constr*s and to insert data transmission semantics, before the *Node*s. Theoretically, if every element in an allocation unit is written before any READ operation, we can set *Constr* to *ConSEQW* or *ConOMPW* and save data transmission. However, it is hard to determine these cases exactly through static analysis. Thus, we set *Constr*s to *ConSEQR*s or *ConOMPR*s to avoid complex static analysis and guarantee the correctness of the programs, regardless of READ and WRITE operations. For an *OMP* and *SEQ*, which is not a function call, we set *Constr* to *ConOMPR* (line 3) and *ConSEQR* (line 13), respectively. For the *SEQ*, which is an *OMP*-called function call (line 6), we set the *Constr* to *ConSEQR*. For the *SEQ*, which is another function call (line 9), we also set the *Constr* to *ConSEQR*, when the corresponding argument is not the pointer or reference type. The reason for this is that the copy of the argument should be guaranteed to be valid before such is passed to the callee function. Then, we insert the data transmission semantic (`OAODataTrans` function), with *ptr* and determined *Constr* as arguments, before the *Node*.

4.3 Updating consistency states

We design Algorithm 5 to determine *TransFunc*s and to insert consistency state update semantics after the *Node*s. When any element of an allocation unit is written in a *Node*, we set the *TransFunc* to *TrSEQW* or *TrOMPW*. READ operations do not change the consistency *State* of *Var*. Thus, all of them are ignored. For an *OMP* and *SEQ*, which is not a function call, we set the *TransFunc* to *TrOMPW* (line 4) and *TrSEQW* (line 14) respectively, if the *RefList* corresponding to *Var* contains any WRITE operation. For the *SEQ*, which is an *OMP*-called function call, we set the *TransFunc* to *TrSEQW* (line 9) if the *RefList* contains any WRITE operation. For the *SEQ*, which is another function call, the essential consistency state update semantics are inserted inside the callee function. Thus, the insertion of state update semantics after the *SEQ* is not needed. Then, we insert the consistency state update semantic (`OAOStTrans` function), with *ptr* and determined *TransFunc* as arguments, after the *Node*.

Algorithm 5: Insert consistency state update semantics

```
Input: $Var$, $Node$
Output: the source code with consistency state update semantics
1  Perform static analysis to get the pointer $ptr$ of $Var$
2  if $Node \in OMPGrp$ then
3      if $\exists W \in RefList$ corresponding to $Var$ then
4          Insert API function OAOStTrans(ptr, TrOMPW) after the $Node$
5      end
6  else if $Node \in SEQGrp \wedge Node$ is a function call then
7      if the callee function is an $OMP$-called function then
8          if $\exists W \in RefList$ corresponding to $Var$ then
9              Insert API function OAOStTrans(ptr, TrSEQW) after the $Node$
10         end
11     end
12 else
13     if $\exists W \in RefList$ corresponding to $Var$ then
14         Insert API function OAOStTrans(ptr, TrSEQW) after the $Node$
15     end
16 end
```

4.4 Parallel primitive translation

Concerning task identification and task mapping, OMP SMP and OMP Offloading both support the work-sharing model well. OMP SMP also supports the task model completely, whereas OMP Offloading only has very limited support for the task model. Thus, this work focuses on the works-haring model.

The corresponding relationships between parallel primitives of OMP SMP and OMP Offloading are listed in Table 7. To exploit GPU, work-sharing loops are distributed across all GPU teams with `teams distribute` primitive. We translate the OMP SMP parallel primitives into the corresponding OMP Offloading parallel primitives according to Table 7. Then, parallel code regions can run

Table 7 OMP parallel primitives and corresponding OMP Offloading parallel primitives

OpenMP parallel primitives	OpenMP offloading parallel primitives
`#pragma omp parallel for`	`#pragma omp target teams distribute parallel for`
`#pragma omp parallel loop`	`#pragma omp target teams distribute parallel loop`
`#pragma omp parallel simd`	`#pragma omp target teams distribute parallel simd`

on the accelerator. The accelerator usually has much more physical cores and threads than CPU. So OAO-translated programs may gain performance improvements. This is summarized as Question 2 in the experimental part and will be verified by experiments.

5 Experimental setup

5.1 Evaluation goals

Our experiments are designed to answer the following questions:

Question 1 *Can OAO translate OMP SMP programs into OMP Offloading programs without manual intervention?*

Question 2 *Can OAO-translated programs gain performance improvements?*

Question 3 *Can OAO optimize data transmission and eliminate redundant transmissions?*

5.2 Benchmarks

The PolyBench [25] and Rodinia [4, 31] are commonly used benchmark suites in the field of high performance computing (HPC). The PolyBench [25] collects many common algorithms in fields such as linear algebra, algebra solvers, data mining, stencils, and image processing. The Rodinia [4, 31] includes some practical applications or kernels such as breadth-first search, computational fluid dynamics, n-body problem, LU decomposition, DNA sequencing, particle filter, and image processing. These fields or applications require the energy-efficient high performance of accelerators. Different types of workloads of the PolyBench and Rodinia can comprehensively evaluate OAO-translated programs. Some related work [18, 30] used the PolyBench or Rodinia in experiments. The OMP SMP version of the PolyBench and Rodinia is suitable as the input of our OAO. So we also use the PolyBench and Rodinia for evaluation.

Table 8 Benchmarks used in experiments

Suite	Benchmark	Description
PolyBench	2DCONV	2-D Convolution
	2MM	2 Matrix Multiplications
	3DCONV	3-D Convolution
	3MM	3 Matrix Multiplications
	ATAX	Matrix Transpose and Vector Multiplication
	BICG	BiCG Sub Kernel of BiCGStab Linear Solver
	CORR	Correlation Computation
	COVAR	Covariance Computation
	FDTD-2D	2-D Finite Different Time Domain Kernel
	FDTD-2D-FUNC	FDTD-2D implemented with subfunctions
	GEMM	Matrix-multiply
	GESUMMV	Scalar, Vector and Matrix Multiplication
	MVT	Matrix Vector Product and Transpose
	SYR2K	Symmetric rank-2k operations
	SYRK	Symmetric rank-k operations
Rodinia	bfs	Breadth-First Search (BFS) algorithm
	cfd_euler	CFD solver with redundant flux computation
	cfd_pre_euler	CFD solver with pre-computed fluxes
	lavaMD	N-Body problem within a large 3D space
	lud	LU Decomposition
	nw	Needleman-Wunsch method for DNA sequencing
	particlefilter	Particle Filter (PF)
	srad_v2	Speckle Reducing Anisotropic Diffusion

We evaluate OAO by applying it to 23 benchmarks from the PolyBench [16] and Rodinia [31] benchmark suites, as listed in Table 8. Moreover, we add a new benchmark called FDTD-2D-FUNC to evaluate data transmission optimizations when programs contain inter-procedural function calls. This benchmark is derived from FDTD-2D with each kernel replaced by a call to the subfunction which encapsulates the kernel. The data required by the kernel are passed as function parameters. We configured all benchmarks in the PolyBench to use single precision for all experiments.

We consider the following five versions of benchmark implementations:

OMP This version refers to the OMP SMP parallel programs from PolyBench and Rodinia. We insert OMP SMP primitives to PolyBench manually, to generate an OMP version PolyBench. Rodinia contains the OMP version natively. OMP version is input and baseline.

OAO This version refers to OMP Offloading programs translated by OAO.

Manual This version refers to OMP Offloading programs translated by hand. The Manual version uses simple copy-in and copy-out data transmission strategy for each offloading kernel. We use `#pragma omp target teams distribute parallel for` directive to offload kernels. We use `cudaMallocHost()` to replace `malloc()`, when the memory block is larger than 128 KB.

DawnCC-native This version refers to origin OMP Offloading programs translated by DawnCC. DawnCC [15] is the state-of-the-art translator that generates OMP Offloading programs. Thus, we use DawnCC for comparison. DawnCC uses `#pragma omp target parallel for` directive to offload kernels.

DawnCC-opt This version refers to DawnCC-native version with our additional optimizations. We introduce two optimizations, which are used in Manual and OAO versions, into DawnCC-opt. We replace `#pragma omp target parallel for` directive with `#pragma omp target teams distribute parallel for` directive. We replace `malloc()` with `cudaMallocHost()` when the memory block is larger than 128 KB. In the comparison of the OAO and DawnCC-opt version, data transmission optimizations of OAO and DawnCC can be evaluated.

5.3 Hardware and software platforms

Table 9 lists the two CPU-GPU systems used in experiments. We use GCC version 8.3 to compile the OMP SMP programs. We use Clang version 9.0 to compile the OMP Offloading programs. CUDA is needed during the compilation and running of the OMP Offloading programs. All compilation processes use optimization level three (-O3).

6 Experimental results

6.1 Performance evaluation

We run each benchmark twenty times and use averages to build the following figures and tables. Figures 5 and 6 show the speedups of different versions over OMP on two CPU-GPU systems. OAO and DawnCC can translate all fifteen benchmarks in PolyBench, whereas only OAO can translate the eight benchmarks in Rodinia. Benchmarks, which DawnCC cannot handle, are marked with ‘X’ in figures.

Table 9 Hardware and Software Platforms

	K40 system	2080Ti system
CPU	2*Intel Xeon E5-2620V3 (6cores/12threads)	2*Intel Xeon E5-2697v4 (18cores/18threads)
CPU Mem	8*16GB DDR3	8*32GB DDR4
GPU	1*K40m	1*RTX 2080Ti
GPU Mem	11GB	11GB
OS	Ubuntu 16.04 (Linux 4.15)	Manjaro 18.1 (Linux 4.19)
Compiler	CUDA-10.1, Clang/LLVM-9.0.0, GCC-8.3.0	CUDA-10.1, Clang/LLVM-9.0.0, GCC-8.3.0

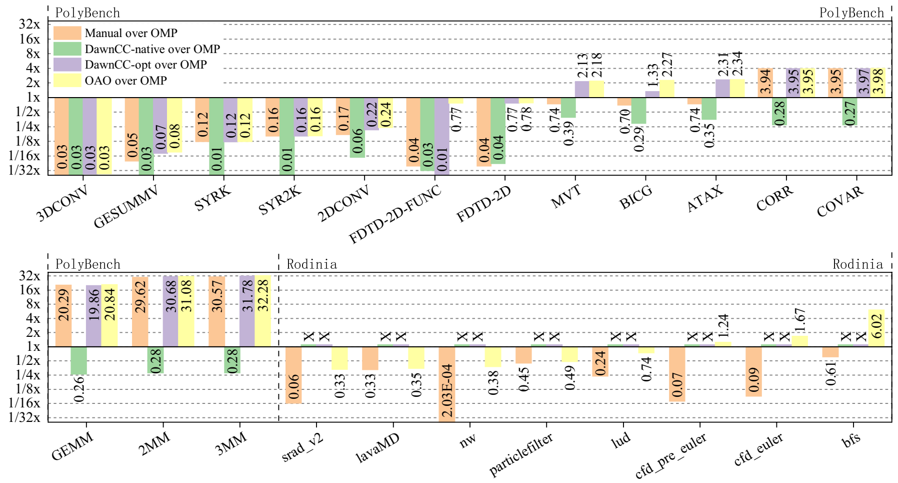

Fig. 5 Speedups over OMP on the K40 system

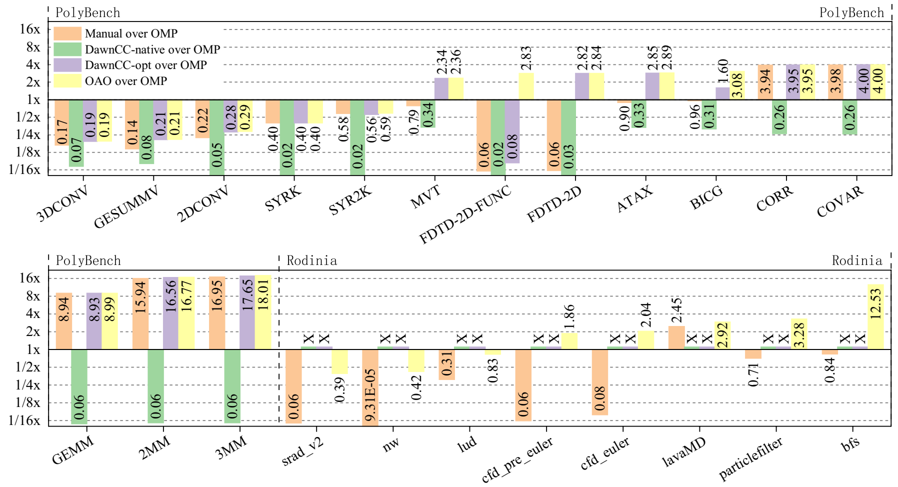

Fig. 6 Speedups over OMP on the RTX system

6.1.1 Performance of OAO version

The OAO version gains performance improvements (1.86× to 32×) over OMP version in more benchmarks than the three other versions, eleven and fifteen benchmarks on K40 and 2080Ti platforms. Besides, the overheads of OAORT are less than 0.07% of the total execution time in all twenty-three benchmarks. The OAO version achieves high speedups in four benchmarks: GEMM, 2MM, 3MM, and bfs.

Speedups of 2MM and 3MM are over 30x. These four benchmarks are compute-intensive applications and suitable for offloading.

The OAO version has poor speedups in eight benchmarks. For 3DCONV, GESUMMV, 2DCONV, nw, and lud, the time of essential data transmissions makes up a large proportion (over 50% to over 90%) of the total execution time of the OAO version. So there is no big chance for data transmissions optimization in these benchmarks. For 3DCONV, SYR2K, GESUMMV, SYRK, and srad_v2, the pure execution time (excluding transmission time and OAORT overhead) of the OAO version is longer than the OMP version. It seems that these applications are not suitable for heterogeneous platforms. To solve this problem, a promising approach is to make the translator can predict application performances on different platforms [6, 32, 35, 36], and automatically decide whether to offload or not.

The OAO version shows different performance on various platforms in FDTD-2D, FDTD-2D-FUNC, lavaMD, and particlefilter. Performance improvements are gained on the 2080Ti platform, whereas poor speedups appear on the K40 platform. The reason is that the more advanced RTX2080Ti GPU can support these benchmarks better.

The performances on two platforms are generally similar. Thus, later discussions and analyses only use data on the 2080Ti system for simplicity.

6.1.2 Comparison with other versions

The OAO version gains performance improvements over three other versions in all benchmarks (Fig. 7). Compared with the Manual version, OAO version achieves large improvements (over 40%) in thirteen benchmarks, and huge improvements (over 500%) in seven benchmarks, especially nw (455,271%). These performance improvements thanks to the data transmission optimization in OAO, which will be analyzed later.

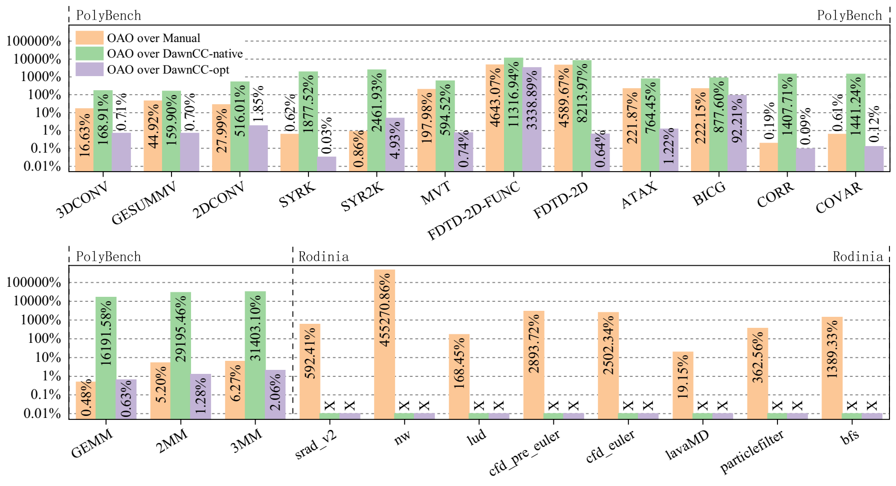

Fig. 7 OAO performance improvements over other versions on the RTX system

DawnCC cannot translate the eight benchmarks in Rodinia correctly because some syntax, such as structure and class, cannot be handled. The OAO version outperforms the DawnCC-native version (over 159%) in all fifteen benchmarks, which DawnCC can handle. Improvements are huge (over 500%) in thirteen of them. Compared with the DawnCC-opt version, the OAO version achieves slight improvements (less than 5%) in thirteen benchmarks. Significant (3339%) and large (92%) improvements are observed in FDTD-2D-FUNC and BICG.

Generally, the OAO version is far better than the DawnCC-native version. However, the OAO version is similar to the DawnCC-opt version in most PolyBench benchmarks. This phenomenon demonstrates that OAO improvements over DawnCC-native are mainly caused by two extra optimizations. The OAO improvements over DawnCC-opt are due to different transmission optimizations in OAO and DawnCC. The time of redundant data transmissions makes up a slight proportion (less than 5%) of the total execution time of the DawnCC-opt version in most PolyBench benchmarks. Thus, most improvements are insignificant. For FDTD-2D-FUNC and BICG, the time of redundant data transmissions makes up large proportions (82% and 45%) and improvements are significant.

In summary, OAO can gain performance improvements over OMP and outperforms DawnCC, which is the state-of-the-art translator.

6.2 Analysis of data transmission optimization

We analyze the number, size, and time of data transmissions to evaluate the data transmission optimization in OAO.

6.2.1 Number of transmissions

Our OAO runtime is designed to eliminate redundant data transmissions to reduce the data communication overhead. Hence, we report the number of data transmissions and use it to quantify how well OAO is in reducing host-accelerator communication overhead. Table 10 shows the number of transmissions in different versions of benchmarks. The DawnCC-native and DawnCC-opt versions have the equal number of transmissions in each corresponding benchmark, so they are expressed as DawnCC in Table 10. Benchmarks, which DawnCC cannot handle, are marked with '–' in Table 10. Comparing The Manual column and OAO column, the reduction of data transmission frequency occurred in all benchmarks. Particularly, the number is reduced by one to five orders of magnitude, in eight benchmarks (italic cells in Table 10): FDTD-2D, FDTD-2D-FUNC, srad_v2, nw, cfd_pre_euler, cfd_euler, particlefilter, and bfs.

Compared with DawnCC column, OAO reduces the number of transmissions in eight benchmarks (bold italic and italic cells). For seven benchmarks (bold italic) except FDTD-2D-FUNC, OAO can eliminate more redundant inter-procedural data transmissions than DawnCC. The FDTD-2D-FUNC benchmark introduces inter-procedural function calls based on the FDTD-2D benchmark. For the FDTD-2D-FUNC benchmark, OAO can eliminate 19,495 more redundant data transmissions than

Table 10 Number of transmissions in different versions of benchmarks

Name	# of transmissions		
	Manual	DawnCC	OAO
3DCONV	**4**	3	3
GESUMMV	**10**	***7***	6
2DCONV	**4**	3	3
SYRK	**8**	3	3
SYR2K	**6**	4	4
MVT	**12**	7	7
ATAX	**12**	***6***	5
FDTD-2D-FUNC	*27,000*	*19,500*	5
FDTD-2D	*27,000*	***7***	5
BICG	**12**	***8***	7
CORR	**8**	6	6
COVAR	**12**	***6***	4
GEMM	**6**	4	4
2MM	**12**	***7***	6
3MM	**18**	***10***	8
srad_v2	*38,912*	–	1033
nw	*16,380*	–	3
lud	**4092**	–	2046
cfd_pre_euler	*224,002*	–	12
cfd_euler	*128,004*	–	9
lavaMD	**10**	–	6
particlefilter	*3534*	–	262
bfs	*216*	–	7

Bold represents the number of transmissions that can be reduced (in the same order of magnitude). Italics represents the number of transmissions that can be significantly reduced (one to five orders of magnitude). Bold italics represents the suboptimal number of transmissions by DawnCC. Underline represents the optimal number of transmissions by OAO

DawnCC. DawnCC can eliminate most redundant transmissions in FDTD-2D but cannot optimize FDTD-2D-FUNC well, which contains massive inter-procedural function calls, whereas OAO can optimize FDTD-2D and FDTD-2D-FUNC to the same minimum number of transmissions (5 times). These phenomena demonstrate that OAO can optimize inter-procedural and intra-procedural data transmissions and outperform DawnCC, which can only optimize intra-procedural data transmissions incompletely.

6.2.2 Data size and time of transmission

Figures 8 and 9 show the percentage of data transmission size and time saved of OAO compared with other versions. The DawnCC-native and DawnCC-opt versions have the equal transmission size in each corresponding benchmark, so they are expressed as DawnCC in Fig. 8.

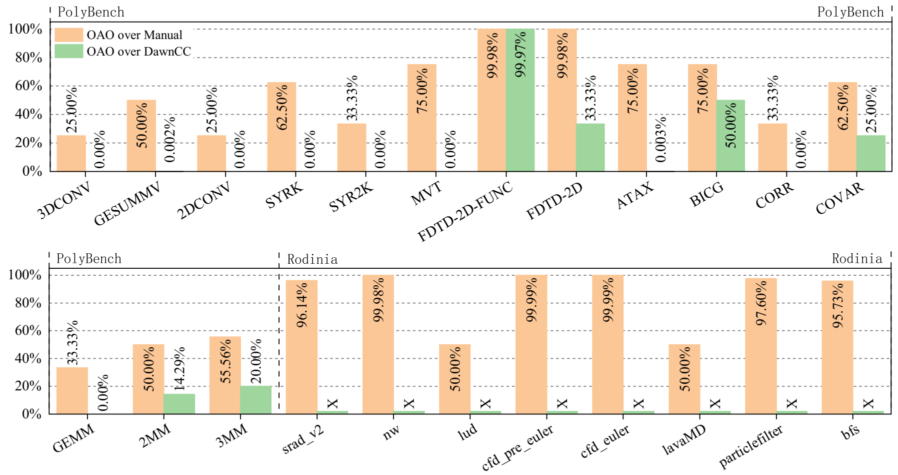

Fig. 8 Percentage of data transmission size saved by OAO compared with other versions

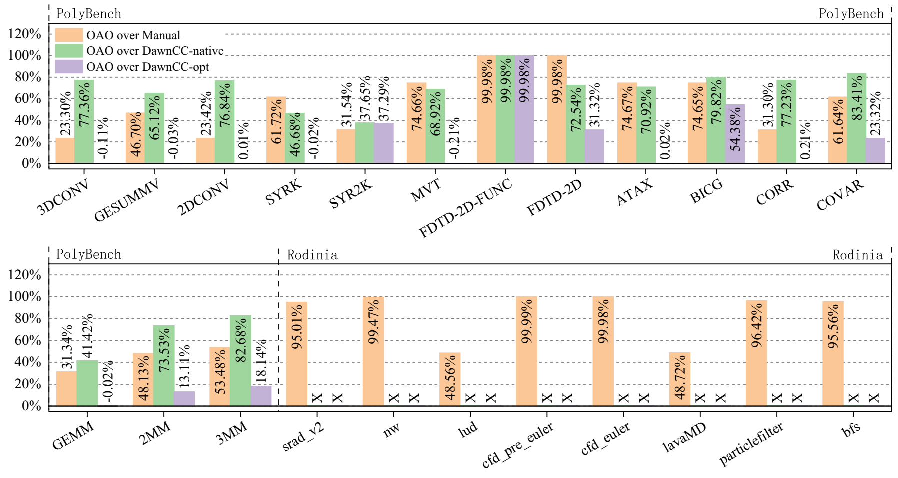

Fig. 9 Percentage of data transmission time saved by OAO compared with other versions on the 2080Ti system

Compared with the Manual version, OAO gains over 25% transmission size savings and over 23% transmission time savings in all benchmarks. For the eighteen benchmarks, these savings are around or more than 50%. For eight benchmarks, these savings are over 95%. Significant performance improvements over Manual version owe to these transmission time savings

The time savings of OAO over Manual are slightly smaller than the corresponding size savings of OAO over Manual. The reason is that OAO eliminates more transmissions from the accelerator to the CPU (D2H) than transmissions from the CPU to the accelerator (H2D). The D2H transmissions usually have higher bandwidth (around 12.25 GB/s) than the H2D transmissions (around 10.55 GB/s). For a block of memory, its D2H time is usually shorter than its H2D time. In a frequent case, its H2D is essential and remained, whereas its D2H is redundant and eliminated. As a consequence, the percentage of data size saved is 50%, whereas the percentage of time saved is less than 50%.

Compared with DawnCC, OAO reduces data transmission size in eight benchmarks. This occurrence matches the transmission number reductions shown in Table 10. The transmission size saving of the six benchmarks is apparent, especially FDTD-2D-FUNC (over 99%), whereas the reduction is negligible in two other benchmarks: GESUMMV and ATAX.

DawnCC-native and DawnCC-opt show different results in terms of transmission time. OAO gains significant reductions (over 41%) on transmission time over DawnCC-native in all benchmarks. For most benchmarks, except FDTD-2D-FUNC, reductions are mainly caused by the pinned memory, which is allocated by **cudaMallocHost()** function and is beneficial to make full use of the bandwidth between the CPU and the accelerator. For FDTD-2D-FUNC, the main reason is that optimization in OAO can eliminate nearly all intra-procedural and inter-procedural transmissions. These significant reductions contribute to huge performance improvements (over 159%), as shown in Fig. 7.

Compared with DawnCC-opt, OAO gains obvious reductions on transmission time in the six benchmarks: FDTD-2D-FUNC, FDTD-2D, BICG, COVAR, 2MM, and 3MM. Combining Figs. 8 and 9, the percentage of time saved and the percentage of the corresponding size saved can match each other in these six benchmarks. Combining Figs. 7 and 9, the six reductions have different contributions to performance improvement, because of diverse proportions of redundant transmission time, as we analyzed before. For FDTD-2D-FUNC and BICG, huge performance improvements (3339% and 92%) thanks to transmission time reductions. For the four other benchmarks, small performance improvements are gained by transmission time reductions.

The above experimental results prove that OAO can optimize data transmission and eliminate redundant transmissions, whether they are intra-procedural or inter-procedural transmissions. OAO can eliminate more redundant transmissions than DawnCC in eight benchmarks, especially inter-procedural transmissions, which cannot be eliminated by DawnCC.

6.3 Implementation and feasibility

We implement the OAO system based on the Clang compiler [13] of the LLVM compiler infrastructure [10, 14] within 14,000 lines of code. The OAO Translator is derived mainly from the `RecursiveASTVisitor` class of Clang. We override about 40 functions of `RecursiveASTVisitor` to carry out compile-time static analysis and translate source code. The OAO Runtime, about 700 lines of code, mainly maintains a vector of variable information and performs Algorithm 2 to execute essential data transmissions. Thus, the implementation complexity of OAO is acceptable.

OAO system can translate the OpenMP code into the OpenMP Offloading code fully automatically. It should be noted that OAO depends on a specific version of the Clang compiler. Developers can follow the detailed instructions on the GitHub to compile and use OAO (https://github.com/ruixueqingyang/OAO-Translator). Thus, OAO is highly feasible to use for developers.

7 Related work

Much work has been exerted to make heterogeneous computing accessible to developers and researchers. Among them, the source-to-source translator is an ideal tool. Moreover, automatic communication management is a main challenge in translation and has been studied widely and exclusively.

7.1 Source-to-source translation for heterogeneous computing

Many translators, such as C-to-CUDA [2], PPCG [30], BONES [18], and OpenMPC [11], generate CUDA programs for heterogeneous computing. Based on the polyhedral model, C-to-CUDA, and PPCG can only parallelize affine code regions in sequential C programs. BONES can parallelize sequential C programs, which are fitted by algorithmic species and skeletons prepared in advance. OpenMPC builds an abstraction of CUDA based on OpenMP and generates CUDA programs automatically. Other translators generate other kinds of heterogeneous parallel programs. Grewe et al. [19] translate OMP SMP into OpenCL. HTrOP [26] generates OpenCL applications from the LLVM bitcode. CU2CL [28] and Kim et al. [9] translate CUDA into OpenCL to achieve portability. Wu et al. [33] propose NoT, a high-level programming method for heterogeneous systems. Then, the NoT application is translated into OpenCL. OpenABLext [34] generates OpenCL from OpenABL, a domain-specific language. DawnCC [15, 17], which is based on the polyhedral model, translates C into OMP Offloading.

Among these translators, DawnCC and our OAO generate OMP Offloading programs, but DawnCC faces serious performance issues. The performance of the DawnCC version is much lower than the OpenMP SMP version in all PolyBench benchmarks because of inefficient data transmission optimization and parallel directives. In FDTD-2D-FUNC, DawnCC and OAO versions display a huge difference in the number of transmissions (19,500 vs. 5). Moreover, DawnCC cannot handle common syntax, such as structure and class.

7.2 Automatic communication management for heterogeneous computing

Many studies concentrate on automatic communication management between the CPU and the accelerator. Semi-automatic techniques [1] can manage data transmissions through Runtime system, but require developers to insert API functions manually.

Fully automatic methods [7, 8, 24] exploit compile-time static analysis techniques to insert Runtime API functions automatically. Our work follows this idea. CGCM [8] is the first fully automatic system for managing and optimizing CPU–GPU communication. DyManD [7], based on CGCM, supports complex data structures, through page protection mechanism and `mmap` function, which may fail. CGCM and DyManD suffer from redundant transmissions. AMM [24] eliminates redundant transmissions, but is bound with X10CUDA [27]. Sousa et al. [29] perform data coherence analysis on OpenCL code and then insert appropriate OpenCL function calls to minimize the number of data coherence operations. Our work also eliminates redundant transmissions, but cannot support multilevel pointers for the safety concern of `mmap()` function.

For seamless data sharing between CPU and GPU, CUDA 6.0 and later versions support unified virtual memory (UVM) [5], where a unified memory address space is shared across the CPUs and GPUs. Li et al. [12] improved OpenMP GPU data management under UVM. These studies are beneficial supplements to this work to support the complex data structure safely and to improve performance further, if the accelerator is specified as NVIDIA hardware. Besides UVM, Castro et al. [3] propose Heterogeneous Transactional Memory (HeTM). HeTM provides programmers with the logical single memory region, shared among the CPUs and GPUs, with support for atomic transactions.

8 Conclusion and future work

This work describes a novel automatic source-to-source translator system, called OAO, to translate OpenMP SMP programs into OpenMP Offloading programs. The OAO consists of the OAO Runtime Library and the OAO Translator. For the OAO Runtime Library, we define the consistency *State*, state transition function, and consistency state constraint to model data transmission operations and variable references. Based on these, we propose the minimum data transmission algorithm to manage and optimize data transmissions automatically and efficiently. For the OAO Translator, we define the *SPGraph* to encode variable reference information. Based on the *SPGraph*, we design some algorithms to insert runtime semantics and translate parallel primitives automatically. We implement the OAO Translator through the compile-time static analysis technology.

Experiments on PolyBench and Rodinia demonstrate that OAO system gains performance improvements over hand-translated (up to 455,271%) and DawnCC-translated (up to 31,403%) OpenMP Offloading programs. The speedup of the OAO version is up to 32.28× over the OpenMP SMP version. The OAO version can save data transmission time (up to over 99%) compared with manual and DawnCC version. Moreover, OAO can handle eight benchmarks in the Rodinia suite, in which DawnCC cannot handle any benchmark. The OAO version gains performance improvement over the OpenMP SMP version in five out of these eight Rodinia benchmarks (up to 12.53x).

The OAO cannot translate some benchmarks of Rodinia correctly, such as the b+tree and heartwall. The main reason is that these benchmarks contain multilevel pointers, which OAO cannot track. Another reason is that some benchmarks contain special OpenMP directives, such as `#pragma omp master`. Besides, tracking consistency at finer granularity may further improve performance. Future work includes extending the OAO system to support more situations and finer-grained optimization.

Acknowledgements This work was supported in part by the National Key Research and Development Program of China (No. 2017YFB0202901), the Key-Area Research and Development Program of Guangdong Province (No. 2019B010136001), the National Natural Science Foundation of China (NSFC) (No. 61672186), and the Shenzhen Technology Research and Development Fund (No. JCYJ20190806143418198). Professor Zhang is the corresponding author.

References

1. Al-Saber N, Kulkarni M (2015) Semcache++: semantics-aware caching for efficient multi-gpu offloading. In: Proceedings of the 29th ACM on International Conference on Supercomputing. ACM, pp 79–88
2. Baskaran MM, Ramanujam J, Sadayappan P (2010) Automatic c-to-cuda code generation for affine programs. In: International Conference on Compiler Construction. Springer, pp 244–263
3. Castro D, Romano P, Ilic A, Khan AM (2019) Hetm: transactional memory for heterogeneous systems. In: 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, pp 232–244
4. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC). IEEE, pp 44–54
5. Corporation N (2019) Cuda toolkit documentation v10.2.89. https://docs.nvidia.com/cuda. Accessed 10 Dec 2019
6. Huang Y, Li D (2017) Performance modeling for optimal data placement on GPU with heterogeneous memory systems. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp 166–177
7. Jablin TB, Jablin JA, Prabhu P, Liu F, August DI (2012) Dynamically managed data for CPU–GPU architectures. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization. ACM, pp 165–174
8. Jablin TB, Prabhu P, Jablin JA, Johnson NP, Beard SR, August DI (2011) Automatic CPU–GPU communication management and optimization. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp 142–151
9. Kim Y, Kim H (2019) Translating cuda to opencl for hardware generation using neural machine translation. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, pp 285–286
10. Lattner C, Adve V (2004) Llvm: a compilation framework for lifelong program analysis and transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, pp 75–86
11. Lee S, Eigenmann R (2010) Openmpc: extended openmp programming and tuning for gpus. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, pp 1–11
12. Li L, Chapman B (2019) 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. ACM, p 51
13. LLVM AT (2020) Clang: a C language family frontend for llvm. http://clang.llvm.org. Accessed 14 Sep 2020
14. LLVM AT (2020) The LLVM compiler infrastructure. http://llvm.org. Accessed 14 Sep 2020

15. Mendonça G, Guimarães B, Alves P, Pereira M, Araújo G, Pereira FMQ (2017) DAWNCC: automatic annotation for data parallelism and offloading. ACM Trans Archit Code Optim (TACO) 14(2):13
16. Mendonça G, Guimarães B, Pereira FMQ (2018) Benchmarks used to evaluate DAWNCC. http://cuda.dcc.ufmg.br/dawn/benchmarks.zip. Accessed 21 Dec 2018
17. Mendonça GSD, Guimaraes BCF, Alves PRO, Pereira FMQ, Pereira MM, Araújo G (2016) Automatic insertion of copy annotation in data-parallel programs. In: 2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, pp 34–41
18. Nugteren C, Corporaal H (2015) Bones: an automatic skeleton-based c-to-cuda compiler for gpus. ACM Trans Arch Code Optim (TACO) 11(4):35
19. O'Boyle MF, Wang Z, Grewe D (2013) Portable mapping of data parallel programs to opencl for heterogeneous systems. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE Computer Society, pp 1–10
20. OpenMP ARB (2019) Openmp application program interface version 3.1. https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf. Accessed 07 Nov 2019
21. OpenMP ARB (2019) Openmp application program interface version 4.0. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf. Accessed 07 Nov 2019
22. OpenMP ARB (2019) Openmp application program interface version 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Accessed 07 Nov 2019
23. OpenMP ARB (2019) Openmp application program interface version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Accessed 07 Nov 2019
24. Pai S, Govindarajan R, Thazhuthaveetil MJ (2012) 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. ACM, pp 33–42
25. Pouchet LN et al (2018) Polybench/c the polyhedral benchmark suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench. Accessed 21 Dec 2018
26. Riebler H, Vaz G, Kenter T, Plessl C (2019) Transparent acceleration for heterogeneous platforms with compilation to opencl. ACM Trans Arch Code Optim (TACO) 16(2):1–26
27. Saraswat V, Bloom B, Peshansky I, Tardieu O, Grove D (2019) The x10 parallel programming language. http://x10-lang.org. Accessed 10 Dec 2019
28. Sathre P, Gardner M, Feng WC (2019) On the portability of CPU-accelerated applications via automated source-to-source translation. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, pp 1–8
29. Sousa R, Pereira M, Pereira FMQ, Araujo G (2019) Data-flow analysis and optimization for data coherence in heterogeneous architectures. J Parallel Distrib Comput 130:126–139
30. Verdoolaege S, Carlos Juega J, Cohen A, Ignacio Gomez J, Tenllado C, Catthoor F (2013) Polyhedral parallel code generation for cuda. ACM Trans Arch Code Optim (TACO) 9(4):54
31. Wang K, Che S, Skadron K (2019) Rodinia: a benchmark suit for heterogeneous computing. http://lava.cs.virginia.edu/Rodinia/download_links.htm. Accessed 23 June 2019
32. Wang X, Huang K, Knoll A, Qian X (2019) A hybrid framework for fast and accurate gpu performance estimation through source-level analysis and trace-based simulation. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 506–518
33. Wu S, Dong X, Zhang X, Zhu Z (2019) Not: a high-level no-threading parallel programming method for heterogeneous systems. J Supercomput 75(7):3810–3841
34. Xiao J, Andelfinger P, Cai W, Richmond P, Knoll A, Eckhoff D (2020) Openablext: an automatic code generation framework for agent-based simulations on CPU–GPU–FPGA heterogeneous platforms. Concurrency and Computation: Practice and Experience p. e5807
35. Zhang W, Cheng AM, Subhlok J (2015) Dwarfcode: a performance prediction tool for parallel applications. IEEE Trans Comput 65(2):495–507
36. Zhang W, Hao M, Snir M (2017) Predicting hpc parallel program performance based on llvm compiler. Cluster Comput 20(2):1179–1192