IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received September 23, 2021, accepted October 30, 2021, date of publication November 2, 2021, date of current version November 8, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3124856

HCE: A Runtime System for Efficiently Supporting Heterogeneous Cooperative Execution

LANJUN WAN[1], WEIHUA ZHENG[2], AND XINPAN YUAN[1]
[1]School of Computer Science, Hunan University of Technology, Zhuzhou 412007, China
[2]College of Electrical and Information Engineering, Hunan University of Technology, Zhuzhou 412007, China

Corresponding author: Lanjun Wan (wanlanjun@hut.edu.cn)

This work was supported in part by the National Natural Science Foundation for Young Scientists of China under Grant 61702177; in part by the Natural Science Foundation of Hunan Province, China, under Grant 2019JJ60048; and in part by the National Natural Science Foundation of China under Grant 61672224.

ABSTRACT Heterogeneous systems with multiple different compute devices have come into common use recently, and the heterogeneity of the compute device is mainly reflected in three aspects: hardware architecture, instruction set architecture, and processing capability. Heterogeneous CPU-accelerator systems have attracted increasing attention especially. To make full use of multiple CPUs and accelerators to execute data-parallel applications, programmers may need to manually map computation and data to all available compute devices, which is tedious, error-prone, and difficult. Especially for some data-parallel applications, the inter-device communication could easily become the performance bottleneck of multi-device co-execution. Therefore, firstly, a runtime system is designed for supporting heterogeneous cooperative execution (HCE) of data-parallel applications, which can help programmers to automatically and efficiently map computation and data to multiple compute devices. Secondly, an incremental data transfer method is designed to avoid redundant data transfers between devices, and a three-way overlapping communication optimization method based on software pipelining is designed to effectively hide the inter-device communication overhead. Based on our previously proposed feedback-based dynamic and elastic task scheduling (FDETS) scheme and asynchronous-based dynamic and elastic task scheduling (ADETS) scheme, the modified FDETS that supports incremental data transfer and the modified ADETS that supports three-way overlapping communication optimization are proposed, which not only can effectively partition and balance the workload among multiple compute devices but also can significantly reduce data transfer overhead between devices. Thirdly, a prototype of the proposed runtime system is implemented, which provides a set of runtime APIs for task scheduling, device management, memory management, and transfer optimization. Our experimental results show that the communication overhead between devices is greatly reduced using the proposed inter-device communication optimization methods and the multi-device co-execution significantly outperforms the best single-device execution.

INDEX TERMS Communication optimization, cooperative execution, data-parallel applications, dynamic scheduling, heterogeneous systems, runtime system.

I. INTRODUCTION

Heterogeneous systems have become increasingly popular in recent years. Some efforts [1]–[8] have been made to fully utilize the available computational resources of a heterogeneous system to execute parallel applications, which demonstrate that the full utilization of all compute devices can result in significant improvements in performance. However, they require programmers to manually manage the task distribution, data transfers, and load balancing between devices, which would be difficult and bring a huge programming burden. Therefore, it is desired to provide an easier and more efficient way to support the multi-device co-execution of parallel applications.

Recently, some heterogeneous parallel programming models and runtime systems have been devoted to make full use of multiple CPUs and accelerators to execute parallel applications on a heterogeneous CPU-accelerator system, such as SKMD [9], CoopCL [10], EngineCL [11], FinePar [12], CoreTSAR [13], StarPU [14], and OmpSs [15]. These heterogeneous parallel programming models and runtime systems

The associate editor coordinating the review of this manuscript and approving it for publication was Laurence T. Yang.

that support CPU-accelerator co-execution can help application programmers automatically map computation and data to multiple CPUs and accelerators. However, the efficient inter-device task scheduling is still a great challenge for the multi-device co-execution.

Some works [16]–[23] have recently concentrated on inter-device task scheduling strategies in heterogeneous CPU-accelerator systems, which statically split work across multiple compute devices before execution or dynamically determine the workload assignment among multiple compute devices during runtime. These task scheduling strategies provide effective workload distribution by maximizing the utilization of all available compute devices and balancing the workload between devices, but most of them do not take into account inter-device communication optimization. For some data-parallel applications, the inter-device communication could easily become the performance bottleneck of multi-device co-execution.

In recent years, many researchers have studied the inter-device communication optimization in heterogeneous CPU-accelerator systems. Gowanlock and Karsin [24] adopted CUDA streams and pinned memory to pipeline data transfers between CPU and GPU for significantly improving the performance of the heterogeneous sorting algorithm. Zheng *et al.* [25] developed a library named HiWayLib to support efficient inter-device data transfers for pipeline programs executing on hybrid CPU-GPU systems, which can avoid duplicated transfers of the overlapped data by employing the method of region-based lazy copy. Li *et al.* [26] proposed the dual buffer rotation four-stage pipelining scheme, which can achieve a good overlap of CPU computation, GPU computation, and CPU-GPU data transfer. Zhang *et al.* [27] developed a GPU-based parallel secure machine learning framework named ParSecureML to boost the efficiency of secure two-party computation. The fine-grained double pipelining technique for overlapping PCI-E data transfer and GPU computing is adopted in ParSecureML to reduce intra-node communication overhead. Tan *et al.* [28] proposed a fine-grained pipelining algorithm to achieve a good overlapped execution of GPU, CPU, PCI-E bus, and IB network, which significantly optimizes the performance of Linpack benchmark running on large-scale hybrid CPU-GPU clusters. The existing researches prove that the pipelining technology can be adopted to effectively reduce the inter-device communication overhead. However, this requires elaborately designing pipeline programs and inter-device task scheduling strategies, and this will become more complicated especially when there are multiple different compute devices in heterogeneous systems.

In our previous work [23], we proposed two inter-device task scheduling strategies to enable the multi-device co-execution of data-parallel applications, including the feedback-based dynamic and elastic task scheduling (FDETS) strategy and the asynchronous-based dynamic and elastic task scheduling (ADETS) strategy. FDETS and ADETS are preferable for data-parallel applications whose computation and data are uniformly distributed and that are non-uniformly distributed, respectively. The detailed descriptions of FDETS and ADETS are given in Section III. Although our previous work can provide efficient inter-device task scheduling for the multi-device co-execution of data-parallel applications which have a smaller inter-device communication overhead, the performance of FDETS and ADETS are not satisfactory in the multi-device co-execution of data-parallel applications which have a larger inter-device communication overhead, and it still requires a significant amount of development effort to implement the multi-device co-execution of data-parallel applications using FDETS and ADETS for programmers. On the basis of the previously proposed FDETS and ADETS, the following extensions are proposed in this paper: (i) the modified FDETS that supports incremental data transfer is proposed, which can keep a good workload balance and avoid redundant data transfers between devices; (ii) the modified ADETS that supports three-way overlapping communication optimization is proposed, which can effectively split work across devices and hide the inter-device communication overhead; (iii) a runtime system named HCE that enables heterogeneous cooperative execution of data-parallel applications is proposed, which can provide a simple and effective way for application programmers to fully exploit multiple compute devices to cooperatively execute data-parallel kernels (i.e., data-parallel for-loops) on a heterogeneous system.

This paper makes the following main contributions:

- A runtime system named HCE is designed for supporting multi-device co-execution of data-parallel kernels on heterogeneous systems, which can help programmers to automatically and efficiently map computation and data to multiple compute devices.
- An incremental data transfer method is designed to avoid redundant data transfers between devices, and the modified FDETS that supports incremental data transfer is proposed.
- A three-way overlapping communication optimization method based on software pipelining is designed to effectively hide the inter-device communication overhead, and the modified ADETS that supports three-way overlapping communication optimization is proposed.
- A prototype of HCE is implemented that targets a heterogeneous system, which provides a set of runtime APIs for task scheduling, device management, memory management, and transfer optimization.

The rest of this paper is organized as follows. Section II presents the overall design of HCE. Section III describes the previous inter-device task scheduling schemes. Section IV discusses the inter-device communication optimization methods. Section V presents the implementation of HCE. Section VI gives the experimental results. Section VII reviews related work. Section VIII concludes the work.

II. OVERALL DESIGN OF HCE

Fig. 1 shows an overview of HCE. Programmers can use the hybrid OpenMP/CUDA/Intel Offload parallel programming

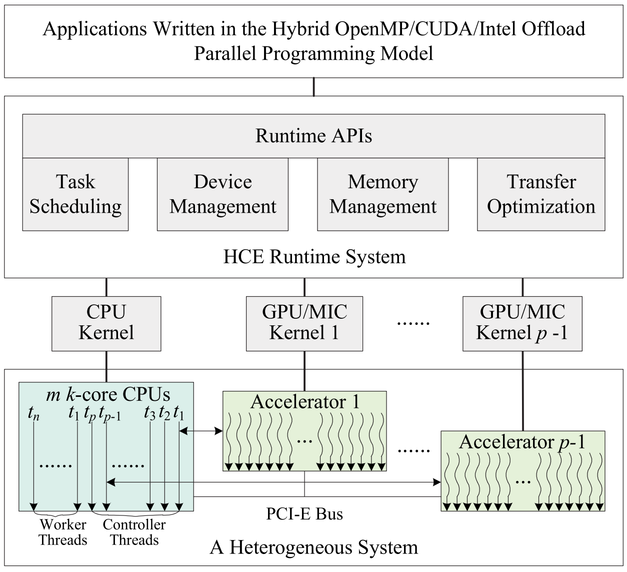


FIGURE 1. An overview of HCE.

model and the runtime APIs provided by HCE to write a program that can be cooperatively executed on multiple devices. Specifically, programmers first identify the computational kernel (i.e., the data-parallel kernel) that needs to be accelerated and determine which compute devices need to participate in multi-device co-execution of the computational kernel. Then, programmers write the device-specific computational kernel for each compute device that participates in multi-device co-execution, such as the CPU/GPU/MIC kernel. Note that the CPU/GPU/MIC kernel is the CPU/GPU/MIC version of the data-parallel code that can run on the CPU/GPU/MIC and is implemented with OpenMP/CUDA/Intel Offload.

As shown in Fig. 1, our HCE runtime system provides some easy-to-use runtime APIs related to task scheduling, device management, memory management, and transfer optimization for application programmers, allowing programmers to make full use of multiple compute devices to cooperatively execute data-parallel applications on a heterogeneous CPU-accelerator system in a simple and effective way. The runtime system is mainly responsible for partitioning and balancing the workload among multiple compute devices, optimizing the inter-device data transfers, and executing the device-specific computational kernel on each compute device to complete its assigned workload. For each computational kernel, it creates the same number of controller threads as the number of compute devices that participate in multi-device co-execution. Specifically, it creates p OpenMP threads to control p compute devices (i.e., $p-1$ many-core accelerators and the multi-core CPUs), where multiple CPUs are seen as one compute device. The thread t_i is in charge of running the device-specific computational kernel on the i-th accelerator, where $1 \leq i \leq p-1$. At first, t_i transfers a part of the input data from the host to the i-th accelerator. Then, t_i launches the available accelerator threads to concurrently perform the computational task assigned to the i-th accelerator. Finally, t_i transfers the results back to the host. At the same time, the thread t_p is in charge of running the CPU kernel on the m k-core CPUs, where m is the number of CPUs and k is the number of cores per CPU. Specifically, we enable the nested parallelism of OpenMP so that t_p spawns the specified number of nested OpenMP threads, called worker threads, to concurrently perform the computational task assigned to the CPUs.

As noted above, we can use multiple CPUs and accelerators to concurrently and cooperatively execute data-parallel kernels. However, the key issue is how to effectively split workload among multiple devices and reduce the inter-device communication overhead, which will be discussed later.

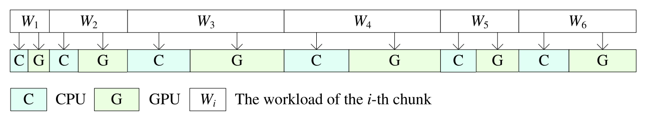
III. PREVIOUS DYNAMIC SCHEDULING SCHEMES

This section briefly describes our previously proposed inter-device task scheduling schemes [23], including the FDETS scheme and the ADETS scheme.

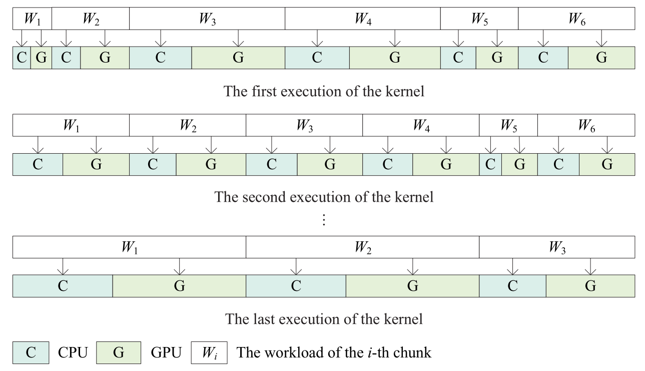
A. THE FDETS SCHEME

FDETS firstly takes $1/n$ of the total workload of a computational kernel (i.e., $1/n$ of the total number of iterations of a data-parallel for-loop) as the initial chunk size and assigns the workload of the initial chunk to each compute device that participates in multi-device co-execution of the kernel according to the initial partition ratios, and then it constantly and dynamically adjusts the chunk size and the partition ratios during execution. Specifically, after the workload of the current chunk has been completed, FDETS dynamically decides whether the next chunk size should be doubled, unchanged or halved compared to the current one according to the performance change of multi-device co-execution, and it dynamically updates the partition ratios that can determine the assignment of the workload of the next chunk between devices by computing the relative execution speed of each compute device.

In order to better understand the FDETS scheme, an example of FDETS is illustrated in Fig. 2. For simplicity, assuming that only a CPU and a GPU are utilized. Fig. 2(a) shows the distribution of workload between the CPU and GPU for a data-parallel kernel to be executed once. As shown in Fig. 2(a), $W_1 = W/16$, $W_2 = 2W_1$, $W_3 = 2W_2$, $W_4 = W_3$, $W_5 = W_4/2$, and $W_6 = W - \sum_{i=1}^{5} W_i$, where W is the total workload of the kernel and W_i is the workload of the i-th chunk. Fig. 2(b) shows the distribution of workload between the CPU and GPU for a data-parallel kernel to be executed several times. As shown in Fig. 2(b), during the 1-th execution of the kernel, $W_1 = W/16$, $W_2 = 2W_1$, $W_3 = 2W_2$, $W_4 = W_3$, $W_5 = W_4/2$, and $W_6 = 3W/16$. Assuming that FDETS finds the 6-th chunk processed at the fastest speed from the 1-th execution of the kernel, the size of the 6-th chunk is used as the sizes of the first two chunks during the 2-th execution of the kernel. As shown in Fig. 2(b), during the 2-th execution of the kernel, $W_1 = 3W/16$, $W_2 = 3W/16$, $W_3 = W_2$, $W_4 = W_3$, $W_5 = W_4/2$, and $W_6 = W - \sum_{i=1}^{5} W_i$.

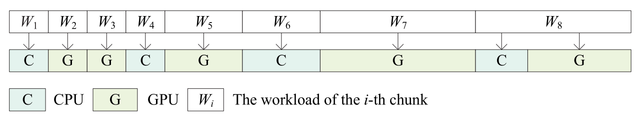

W_1 W_2 W_3 W_4 W_5 W_6
C G C G C G C G C G C G
C CPU G GPU W_i The workload of the i-th chunk

(a) The distribution of workload for a data-parallel kernel to be executed once.

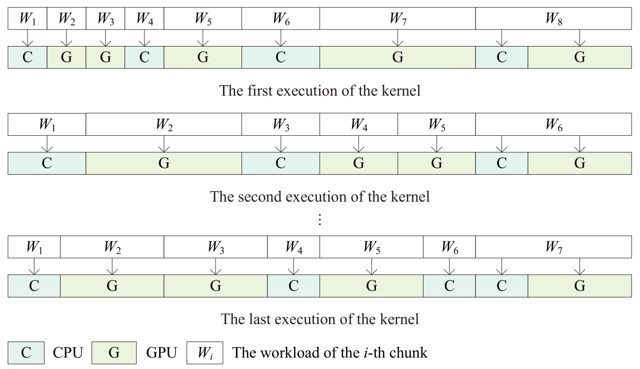

The first execution of the kernel
The second execution of the kernel
The last execution of the kernel
C CPU G GPU W_i The workload of the i-th chunk

(b) The distribution of workload for a data-parallel kernel to be executed many times.

FIGURE 2. An example of FDETS.


W_1 W_2 W_3 W_4 W_5 W_6 W_7 W_8
C G G C G C G C G
C CPU G GPU W_i The workload of the i-th chunk

(a) The distribution of workload for a data-parallel kernel to be executed once.


The first execution of the kernel
The second execution of the kernel
The last execution of the kernel
C CPU G GPU W_i The workload of the i-th chunk

(b) The distribution of workload for a data-parallel kernel to be executed many times.

FIGURE 3. An example of ADETS.

During the last execution of the kernel, $W_1 = 6W/16$, $W_2 = 6W/16$, and $W_3 = W - \sum_{i=1}^{2} W_i$. It also can be seen from Fig. 2 that the partition ratios used to determine the assignment of the workload of one chunk between the CPU and GPU are updated continuously. For example, after the second chunk depicted in Fig. 2(a) has finished processing, the partition ratio of the CPU is updated from 37.1% to 39.8%, while the partition ratio of the GPU is updated from 62.9% to 60.2%.

B. THE ADETS SCHEME

ADETS firstly assigns a chunk whose size is W/n to each compute device that participates in multi-device co-execution of a data-parallel for-loop, and then immediately assigns the next chunk to one compute device once it has completed its work. The size of the next chunk assigned to device D_i is dynamically adjusted according to the current chunk size and the variance between the previous and current execution speeds of device D_i.

Fig. 3 shows an example of ADETS. As shown in Fig. 3, the first and second chunks are assigned to the CPU and GPU respectively, once the CPU or GPU has finished its work, the next unassigned chunk is assigned to it immediately. As shown in Fig. 3(a), the 1-th, 4-th, and 6-th chunks are assigned to the CPU, where $W_1 = W/16$, $W_4 = W/16$, and $W_6 = 2W_4$; the 2-th, 3-th, 5-th, and 7-th chunks are assigned to the GPU, where $W_2 = W/16$, $W_3 = W/16$, $W_5 = 2W_3$, and $W_7 = 2W_5$; the last chunk is assigned to the CPU and GPU according to the partition ratios computed in the previous executions. In Fig. 3(b), we can see that the data-parallel kernel needs to be executed many times, begin from the second execution of the kernel, the sizes of the first two chunks assigned to device D_i are determined by the size of the chunk processed by device D_i at the fastest speed during the previous execution of the kernel. For example, for the CPU, ADETS finds the 6-th chunk processed at the fastest speed from the first execution of the kernel, thus the sizes of the first two chunks assigned to the CPU are all $W/8$ during the second execution of the kernel.

IV. INTER-DEVICE COMMUNICATION OPTIMIZATION

This section describes our proposed two inter-device communication optimization methods.

A. MOTIVATION

If the inter-device task scheduling decision is made without considering data transfer cost on a heterogeneous system, for some data-parallel kernels, the huge inter-device communication overhead would significantly degrade the overall performance of multi-device co-execution, so the inter-device communication can easily become the performance bottleneck of multi-device co-execution. If the data transfer overhead is higher than the performance gain actually achieved by offloading computation, the performance of multi-device co-execution will be worse than that of the best single-device multi-thread parallel execution. If the partitioning decision is made without considering data transfer cost and performance variance of partitioning, it will be suboptimal or even cause slowdown compared to the single-device execution.

As shown in Fig. 4(a), the execution times of the CPU-GPU-MIC co-execution using two different inter-device task scheduling schemes are more than the execution time of the GPU-only execution for Jacobi with three different problem sizes. Fig. 4(b) also shows that the performance of the CPU-GPU-MIC co-execution is not as good as that of the GPU-only or MIC-only execution for FDTD2d with three different problem sizes. The huge CPU-GPU and CPU-MIC communication overheads have a great impact on the overall

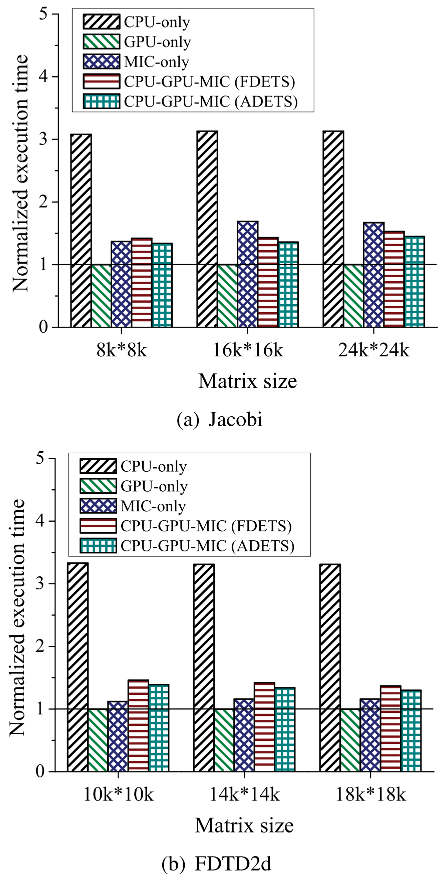
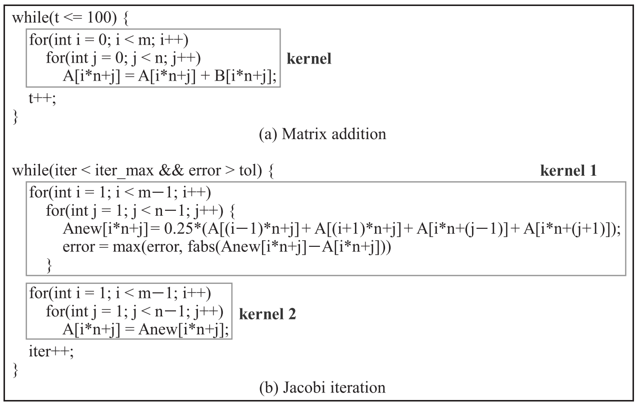


FIGURE 4. The execution time comparison among five different parallel implementations for two different benchmarks.


```
while(t <= 100) {
  for(int i = 0; i < m; i++)
    for(int j = 0; j < n; j++)                       kernel
      A[i*n+j] = A[i*n+j] + B[i*n+j];
  t++;
}
                    (a) Matrix addition

while(iter < iter_max && error > tol) {                                kernel 1
  for(int i = 1; i < m−1; i++)
    for(int j = 1; j < n−1; j++) {
      Anew[i*n+j]= 0.25*(A[(i−1)*n+j]+ A[(i+1)*n+j]+ A[i*n+(j−1)]+ A[i*n+(j+1)]);
      error = max(error, fabs(Anew[i*n+j]−A[i*n+j]))
    }
  for(int i = 1; i < m−1; i++)
    for(int j = 1; j < n−1; j++)   kernel 2
      A[i*n+j] = Anew[i*n+j];
  iter++;
}
                    (b) Jacobi iteration
```

FIGURE 5. Two typical applications that have repeated data transfers caused by multi-device co-execution.

performance of CPU-GPU-MIC co-execution for Jacobi and FDTD2d. Therefore, the inter-device data transfers can easily become the performance bottleneck of multi-device co-execution for some data-parallel kernels, there is a need to do inter-device communication optimization, and especially our proposed task scheduling schemes should take this into account.

B. THE INCREMENTAL DATA TRANSFER METHOD

In this subsection, we first discuss the inter-device redundant transfers and then present the modified FDETS that supports incremental data transfer.

1) THE INTER-DEVICE REDUNDANT TRANSFERS

For the multi-device co-execution of some data-parallel kernels, the inter-device task scheduling may incur large communication costs due to frequent inter-device data transfers. As shown in Fig. 5, the matrix addition contains a computational kernel that needs to be executed repeatedly. For simplicity, we assume that only a CPU and a GPU are used to cooperatively execute the kernel. During each execution of the kernel, a part of array *A* needs to be uploaded from the host to the GPU and downloaded from the GPU to the host due to the change in partition ratios. Similarly, the Jacobi iteration has two computational kernels that need to be executed repeatedly. During each execution of kernel 1, *A* needs to be partially uploaded to the GPU, and *Anew* needs to be partially downloaded from the GPU. During each execution of kernel 2, *Anew* needs to be partially uploaded to the GPU, and *A* needs to be partially downloaded from the GPU.

It is apparent that there are a large amount of inter-device data transfers during the repeated executions of the above-described two applications, which may contain a great deal of redundant transfers. For each accelerator that participates in multi-device co-execution, if the data to be processed on the accelerator in the next execution are already present in the accelerator memory, but the data are downloaded from the accelerator at the end of the current execution and are still uploaded to the accelerator at the beginning of the next execution, such data transfers are considered to be redundant.

2) THE MODIFIED FDETS THAT SUPPORTS INCREMENTAL DATA TRANSFER

To avoid redundant transfers, we design an incremental data transfer method for data-parallel applications which have one or more computational kernels that need to be executed repeatedly, such as the two applications depicted in Fig. 5. To better support the incremental data transfer, we make some modifications to FDETS. Simply put, at the begin of each execution of a computational kernel, the total workload of the kernel is split according to the suitable partition ratios, and we assign a part of the entire workload to the specified compute device. After each compute device has completed its work, we obtain the execution time of each compute device to calculate the new partition ratios.

The key issues to be solved for the incremental data transfer are as follows: (i) how to identify which parts of an array must be uploaded from the host to the specified accelerator at the begin of each execution of a computational kernel; (ii) how to identify which parts of an array must be downloaded from the specified accelerator to the host at the end of each execution of a computational kernel.

Algorithm 1 describes how to determine which parts of an array need to be uploaded from the host to the specified accelerator. Specifically, according to the total number of iterations

Algorithm 1 Determine Which Parts of an Array Need to Be Uploaded From the Host to the Specified Accelerator

```
Require: $p$, $W$, $i$, $R_{\text{prev}.j}$ and $R_{\text{curr}.j}$ ($j = 1$ to $i$)
Initialize $Up_{i.1}^{\text{begin}} = Up_{i.1}^{\text{end}} = Up_{i.2}^{\text{begin}} = Up_{i.2}^{\text{end}} = -1$;
$W_{\text{prev}.i}^{\text{begin}} = W \times \sum_{j=1}^{i-1} R_{\text{prev}.j}$;
$W_{\text{prev}.i}^{\text{end}} = W_{\text{prev}.i}^{\text{begin}} + W \times R_{\text{prev}.i} - 1$;
$W_{\text{curr}.i}^{\text{begin}} = W \times \sum_{j=1}^{i-1} R_{\text{curr}.j}$;
$W_{\text{curr}.i}^{\text{end}} = W_{\text{curr}.i}^{\text{begin}} + W \times R_{\text{curr}.i} - 1$;
if $W_{\text{curr}.i}^{\text{begin}} < W_{\text{prev}.i}^{\text{begin}}$ then
    if $W_{\text{curr}.i}^{\text{end}} < W_{\text{prev}.i}^{\text{begin}}$ then $\triangleright$ Case 1
        $Up_{i.1}^{\text{begin}} = W_{\text{curr}.i}^{\text{begin}}$;
        $Up_{i.1}^{\text{end}} = W_{\text{curr}.i}^{\text{end}}$;
    end if
    if $W_{\text{curr}.i}^{\text{end}} \geq W_{\text{prev}.i}^{\text{begin}}$ and $W_{\text{curr}.i}^{\text{end}} \leq W_{\text{prev}.i}^{\text{end}}$ then $\triangleright$ Case 2
        $Up_{i.1}^{\text{begin}} = W_{\text{curr}.i}^{\text{begin}}$;
        $Up_{i.1}^{\text{end}} = W_{\text{prev}.i}^{\text{begin}} - 1$;
    end if
    if $W_{\text{curr}.i}^{\text{end}} > W_{\text{prev}.i}^{\text{end}}$ then $\triangleright$ Case 3
        $Up_{i.1}^{\text{begin}} = W_{\text{curr}.i}^{\text{begin}}$;
        $Up_{i.1}^{\text{end}} = W_{\text{prev}.i}^{\text{begin}} - 1$;
        $Up_{i.2}^{\text{begin}} = W_{\text{prev}.i}^{\text{end}} + 1$;
        $Up_{i.2}^{\text{end}} = W_{\text{curr}.i}^{\text{end}}$;
    end if
end if
if $W_{\text{curr}.i}^{\text{begin}} \geq W_{\text{prev}.i}^{\text{begin}}$ and $W_{\text{curr}.i}^{\text{begin}} \leq W_{\text{prev}.i}^{\text{end}}$ then
    if $W_{\text{curr}.i}^{\text{end}} \leq W_{\text{prev}.i}^{\text{end}}$ then $\triangleright$ Case 4
        There is no need to upload data to device $D_i$;
    end if
    if $W_{\text{curr}.i}^{\text{end}} > W_{\text{prev}.i}^{\text{end}}$ then $\triangleright$ Case 5
        $Up_{i.1}^{\text{begin}} = W_{\text{prev}.i}^{\text{end}} + 1$;
        $Up_{i.1}^{\text{end}} = W_{\text{curr}.i}^{\text{end}}$;
    end if
end if
if $W_{\text{curr}.i}^{\text{begin}} > W_{\text{prev}.i}^{\text{end}}$ then $\triangleright$ Case 6
    $Up_{i.1}^{\text{begin}} = W_{\text{curr}.i}^{\text{begin}}$;
    $Up_{i.1}^{\text{end}} = W_{\text{curr}.i}^{\text{end}}$;
end if
return $Up_{i.1}^{\text{begin}}$, $Up_{i.1}^{\text{end}}$, $Up_{i.2}^{\text{begin}}$, and $Up_{i.2}^{\text{end}}$
```

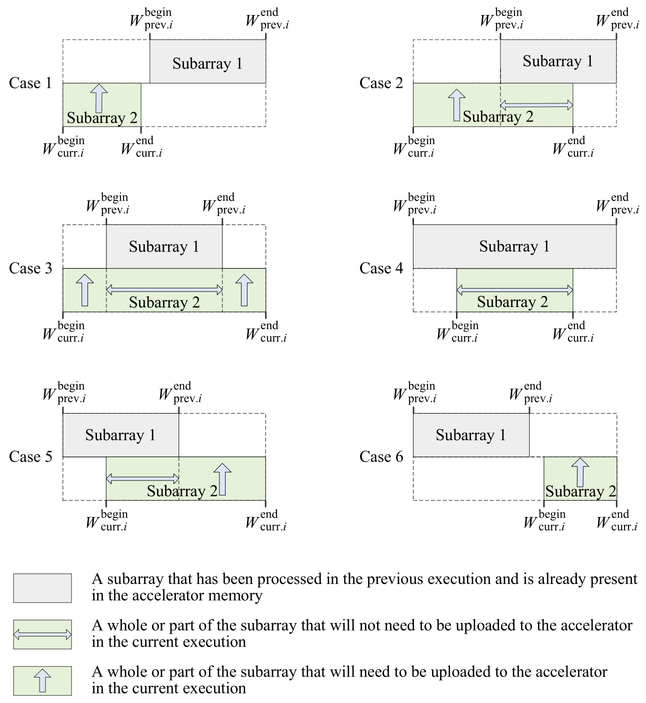

Case 1
Case 2
Case 3
Case 4
Case 5
Case 6
Subarray 1
Subarray 2
$W_{\text{prev}.i}^{\text{begin}}$
$W_{\text{prev}.i}^{\text{end}}$
$W_{\text{curr}.i}^{\text{begin}}$
$W_{\text{curr}.i}^{\text{end}}$
A subarray that has been processed in the previous execution and is already present in the accelerator memory
A whole or part of the subarray that will not need to be uploaded to the accelerator in the current execution
A whole or part of the subarray that will need to be uploaded to the accelerator in the current execution

FIGURE 6. Different scenarios of the incremental data transfer in uploading data from the host to the accelerator.

W of the outermost for-loop of the computational kernel, the previous partition ratios $R_{\text{prev}.1}, R_{\text{prev}.2}, \ldots, R_{\text{prev}.i}$, and the current partition ratios $R_{\text{curr}.1}, R_{\text{curr}.2}, \ldots, R_{\text{curr}.i}$, we firstly get the begin index $W_{\text{prev}.i}^{\text{begin}}$ and the end index $W_{\text{prev}.i}^{\text{end}}$ of a subarray (i.e., a section of an array) that has been processed on the i-th compute device (i.e., a specified accelerator) in the previous execution of the kernel, where $1 \leq i \leq p$ and p is the number of compute devices that participate in multi-device co-execution. Secondly, we get the begin index $W_{\text{curr}.i}^{\text{begin}}$ and the end index $W_{\text{curr}.i}^{\text{end}}$ of a subarray that will need to be processed on the specified accelerator in the current execution of the kernel. Thirdly, we determine which parts of the subarray need to be uploaded to the specified accelerator by comparing $W_{\text{prev}.i}^{\text{begin}}$, $W_{\text{prev}.i}^{\text{end}}$, $W_{\text{curr}.i}^{\text{begin}}$, and $W_{\text{curr}.i}^{\text{end}}$. If the whole subarray to be processed is not present in the accelerator memory, then it needs to be uploaded to the accelerator (see cases 1 and 6 in Fig. 6); if only a part of the subarray to be processed are already present in the accelerator memory, then the other part need to be uploaded to the accelerator (see cases 2, 3, and 5 in Fig. 6); if the whole subarray to be processed is present in the accelerator memory, then it does not need to be uploaded to the accelerator (see case 4 in Fig. 6). Finally, $Up_{i.1}^{\text{begin}}$ and $Up_{i.1}^{\text{end}}$ are used to store the begin index and the end index of the first part of data that need to be uploaded to the accelerator respectively, and $Up_{i.2}^{\text{begin}}$ and $Up_{i.2}^{\text{end}}$ are used to store the begin index and the end index of the second part of data that need to be uploaded to the accelerator respectively.

Algorithm 2 describes how to determine which parts of an array need to be downloaded from the specified accelerator to the host. Similar to Algorithm 1, according to the total number of iterations W, the current partition ratios $R_{\text{curr}.1}$, $R_{\text{curr}.2}, \ldots, R_{\text{curr}.i}$, and the next partition ratios $R_{\text{next}.1}, R_{\text{next}.2}$, $\ldots, R_{\text{next}.i}$, we firstly get the begin index $W_{\text{curr}.i}^{\text{begin}}$ and the end index $W_{\text{curr}.i}^{\text{end}}$ of a subarray that has been processed on the i-th compute device (i.e., a specified accelerator) in the current execution of the kernel, where $1 \leq i \leq p$. Secondly, we get the begin index $W_{\text{next}.i}^{\text{begin}}$ and the end index $W_{\text{next}.i}^{\text{end}}$ of a subarray that will need to be processed on the specified accelerator in the next execution of the kernel. Thirdly, we determine

Algorithm 2 Determine Which Parts of an Array Need to Be Downloaded From the Specified Accelerator to the Host

```
Require: $p$, $W$, $i$, $R_{\text{curr}.j}$ and $R_{\text{next}.j}$ ($j = 1$ to $p$)
1:  Initialize $Down_{i.1}^{\text{begin}} = Down_{i.1}^{\text{end}} = Down_{i.2}^{\text{begin}} = Down_{i.2}^{\text{end}} = -1$;
2:  $W_{\text{curr}.i}^{\text{begin}} = W \times \sum_{j=1}^{i-1} R_{\text{curr}.j}$;
3:  $W_{\text{curr}.i}^{\text{end}} = W_{\text{curr}.i}^{\text{begin}} + W \times R_{\text{curr}.i} - 1$;
4:  $W_{\text{next}.i}^{\text{begin}} = W \times \sum_{j=1}^{i-1} R_{\text{next}.j}$;
5:  $W_{\text{next}.i}^{\text{end}} = W_{\text{next}.i}^{\text{begin}} + W \times R_{\text{next}.i} - 1$;
6:  if $W_{\text{next}.i}^{\text{begin}} \leq W_{\text{curr}.i}^{\text{begin}}$ then
7:      if $W_{\text{next}.i}^{\text{end}} < W_{\text{curr}.i}^{\text{begin}}$ then ▷ Case 1
8:          $Down_{i.1}^{\text{begin}} = W_{\text{curr}.i}^{\text{begin}}$;
9:          $Down_{i.1}^{\text{end}} = W_{\text{curr}.i}^{\text{end}}$;
10:     end if
11:     if $W_{\text{next}.i}^{\text{end}} \geq W_{\text{curr}.i}^{\text{begin}}$ and $W_{\text{next}.i}^{\text{end}} < W_{\text{curr}.i}^{\text{end}}$ then ▷ Case 2
12:         $Down_{i.1}^{\text{begin}} = W_{\text{next}.i}^{\text{end}} + 1$;
13:         $Down_{i.1}^{\text{end}} = W_{\text{curr}.i}^{\text{end}}$;
14:     end if
15:     if $W_{\text{next}.i}^{\text{end}} \geq W_{\text{curr}.i}^{\text{end}}$ then ▷ Case 3
16:         There is no need to download data from device $D_i$;
17:     end if
18: end if
19: if $W_{\text{next}.i}^{\text{begin}} > W_{\text{curr}.i}^{\text{begin}}$ and $W_{\text{next}.i}^{\text{begin}} \leq W_{\text{curr}.i}^{\text{end}}$ then
20:     if $W_{\text{next}.i}^{\text{end}} < W_{\text{curr}.i}^{\text{end}}$ then ▷ Case 4
21:         $Down_{i.1}^{\text{begin}} = W_{\text{curr}.i}^{\text{begin}}$;
22:         $Down_{i.1}^{\text{end}} = W_{\text{next}.i}^{\text{begin}} - 1$;
23:         $Down_{i.2}^{\text{begin}} = W_{\text{next}.i}^{\text{end}} + 1$;
24:         $Down_{i.2}^{\text{end}} = W_{\text{curr}.i}^{\text{end}}$;
25:     end if
26:     if $W_{\text{next}.i}^{\text{end}} \geq W_{\text{curr}.i}^{\text{end}}$ then ▷ Case 5
27:         $Down_{i.1}^{\text{begin}} = W_{\text{curr}.i}^{\text{begin}}$;
28:         $Down_{i.1}^{\text{end}} = W_{\text{next}.i}^{\text{begin}} - 1$;
29:     end if
30: end if
31: if $W_{\text{next}.i}^{\text{begin}} > W_{\text{curr}.i}^{\text{end}}$ then ▷ Case 6
32:     $Down_{i.1}^{\text{begin}} = W_{\text{curr}.i}^{\text{begin}}$;
33:     $Down_{i.1}^{\text{end}} = W_{\text{curr}.i}^{\text{end}}$;
34: end if
35: return $Down_{i.1}^{\text{begin}}$, $Down_{i.1}^{\text{end}}$, $Down_{i.2}^{\text{begin}}$, and $Down_{i.2}^{\text{end}}$
```

which parts of the subarray need to be downloaded from the specified accelerator by comparing $W_{\text{curr}.i}^{\text{begin}}$, $W_{\text{curr}.i}^{\text{end}}$, $W_{\text{next}.i}^{\text{begin}}$, and $W_{\text{next}.i}^{\text{end}}$. If the whole subarray updated on the accelerator in the current execution is not need for the accelerator in the next execution, then it needs to be downloaded from the accelerator (see cases 1 and 6 in Fig. 7); if only a part of the subarray updated on the accelerator in the current execution are need for the accelerator in the next execution, then the

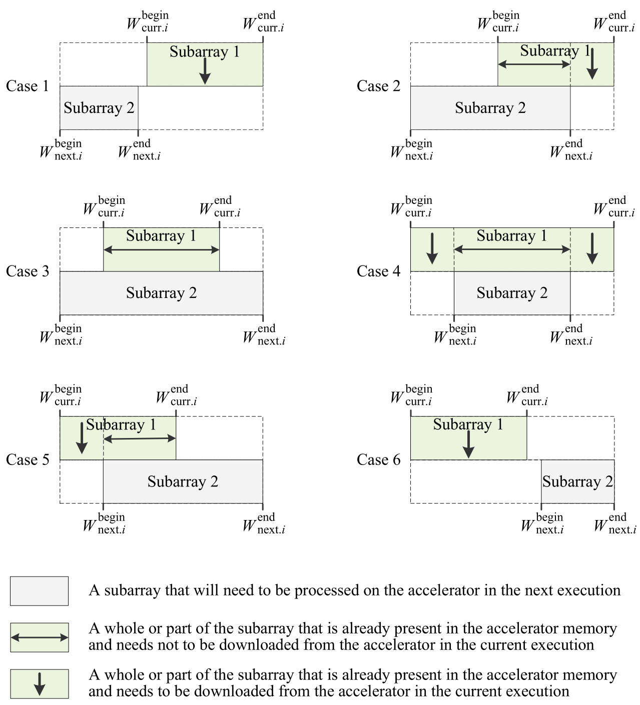

Case 1
Subarray 1
Subarray 2
Case 2
Case 3
Case 4
Case 5
Case 6
$W_{\text{curr}.i}^{\text{begin}}$
$W_{\text{curr}.i}^{\text{end}}$
$W_{\text{next}.i}^{\text{begin}}$
$W_{\text{next}.i}^{\text{end}}$
A subarray that will need to be processed on the accelerator in the next execution
A whole or part of the subarray that is already present in the accelerator memory and needs not to be downloaded from the accelerator in the current execution
A whole or part of the subarray that is already present in the accelerator memory and needs to be downloaded from the accelerator in the current execution

FIGURE 7. **Different scenarios of the incremental data transfer in downloading data from the accelerator to the host.**

other part need to be downloaded from the accelerator (see cases 2, 4, and 5 in Fig. 7); if the whole subarray updated on the accelerator in the current execution is need for the accelerator in the next execution, then it does not need to be downloaded from the accelerator (see case 3 in Fig. 7). Finally, $Down_{i.1}^{\text{begin}}$ and $Down_{i.1}^{\text{end}}$ are used to store the begin index and the end index of the first part of data that need to be downloaded from the accelerator respectively, and $Down_{i.2}^{\text{begin}}$ and $Down_{i.2}^{\text{end}}$ are used to store the begin index and the end index of the second part of data that need to be downloaded from the accelerator respectively.

Algorithm 3 describes the modified FDETS that supports incremental data transfer. Supposing that a computational kernel needs to be executed *tolExecs* times repeatedly. The incremental data transfer can be considered in each execution of the kernel, except for uploading data from the host to the accelerator in the first execution and downloading data from the accelerator to the host in the last execution. Moreover, the partition ratios that will be used in the next execution of the kernel need to be determined in advance when considering the incremental data transfer. Specifically, the initial partition ratios are adopted in the first and second executions of the kernel. Starting with the second execution of the kernel, the partition ratios that will be used in the next execution are determined by the new partition ratios computed in the previous execution.

As seen in Algorithm 3, during each execution of the computational kernel, we firstly assign the workload $W_{\text{curr}.i}$ (i.e., a part of the total workload W) to device D_i according to its current partition ratio $R_{\text{curr}.i}$, where $1 \leq i \leq p$. Secondly,

Algorithm 3 The Modified FDETS That Supports Incremental Data Transfer

```
Require: $p$, $W$, the initial partition ratios $R_1, R_2, \ldots, R_p$, and $tolExecs$
1: Initialize $R_{\text{prev}.i} = 0$ and $R_{\text{curr}.i} = R_{\text{next}.i} = R_i$ ($i = 1$ to $p$);
2: for $t = 1$ to $tolExecs$ do
3:     for each controller thread $t_i$, $1 \leq i \leq p$, in parallel do
4:         Assign the workload $W_{\text{curr}.i} = W \times R_{\text{curr}.i}$ to $D_i$;
5:         if device $D_i$ is an accelerator then
6:             Perform Algorithm 1 to get $Up_{i.1}^{\text{begin}}$, $Up_{i.1}^{\text{end}}$, $Up_{i.2}^{\text{begin}}$, and $Up_{i.2}^{\text{end}}$;
7:             Upload the data indexed from $Up_{i.1}^{\text{begin}}$ to $Up_{i.1}^{\text{end}}$ to device $D_i$ when $Up_{i.1}^{\text{begin}} \geq 0$;
8:             Upload the data indexed from $Up_{i.2}^{\text{begin}}$ to $Up_{i.2}^{\text{end}}$ to device $D_i$ when $Up_{i.2}^{\text{begin}} \geq 0$;
9:         end if
10:        Execute the kernel on $D_i$ to complete $W_{\text{curr}.i}$;
11:        if device $D_i$ is an accelerator then
12:            Perform Algorithm 2 to get $Down_{i.1}^{\text{begin}}$, $Down_{i.1}^{\text{end}}$, $Down_{i.2}^{\text{begin}}$, and $Down_{i.2}^{\text{end}}$;
13:            Download the data indexed from $Down_{i.1}^{\text{begin}}$ to $Down_{i.1}^{\text{end}}$ from device $D_i$ when $Down_{i.1}^{\text{begin}} \geq 0$;
14:            Download the data indexed from $Down_{i.2}^{\text{begin}}$ to $Down_{i.2}^{\text{end}}$ from device $D_i$ when $Down_{i.2}^{\text{begin}} \geq 0$;
15:        end if
16:        Obtain the execution time $T_{\text{curr}.i}$ of device $D_i$;
17:        Compute the execution speed of device $D_i$: $V_{\text{curr}.i} = W_{\text{curr}.i}/T_{\text{curr}.i}$;
18:        Update the partition ratios of device $D_i$: $R_{\text{prev}.i} = R_{\text{curr}.i}$, $R_{\text{curr}.i} = R_{\text{next}.i}$;
19:    end for
20:    if $t < tolExecs - 1$ then
21:        $R_{\text{next}.i} = V_{\text{curr}.i}/\sum_{j=1}^{p} V_{\text{curr}.j}$ ($i = 1$ to $p$);
22:    else
23:        $R_{\text{next}.i} = 0$ ($i = 1$ to $p$);
24:    end if
25: end for
```

we use device D_i to execute the kernel to complete $W_{\text{curr}.i}$. If device D_i is an accelerator, we perform Algorithm 1 to identify which parts of the data need to be uploaded from the host to the accelerator before execution and copy these data to the accelerator memory, and we perform Algorithm 2 to identify which parts of the data need to be downloaded from the accelerator to the host after execution and copy these data to the host memory. Thirdly, we obtain the current execution time $T_{\text{curr}.i}$ of device D_i to compute its current execution speed $V_{\text{curr}.i}$. If device D_i is an accelerator, the current execution time should include the data transfer time. Fourthly, we update the previous and current partition ratios of device D_i: $R_{\text{prev}.i} = R_{\text{curr}.i}$ and $R_{\text{curr}.i} = R_{\text{next}.i}$. Finally, after all p compute devices have completed the cooperative execution of the kernel, we update the next partition ratio of device D_i: $R_{\text{next}.i} = V_{\text{curr}.i}/\sum_{j=1}^{p} V_{\text{curr}.j}$.

It is easy to see that the time complexity of both Algorithm 1 and Algorithm 2 is $O(2p)$ in the worst case, where p is the number of compute devices that participate in multi-device co-execution. As shown in Algorithm 3, if one compute device is an accelerator, Algorithm 1 needs to be executed to determine which parts of an array will to be uploaded from the host to the accelerator, and Algorithm 2 also needs to be executed to determine which parts of an array will to be downloaded from the accelerator to the host. Therefore, the time complexity of Algorithm 3 is $O(7\lambda p)$ in the worst case, where λ is the number of times the computational kernel needs to be executed repeatedly. Thus it can be seen that the modified FDETS that supports incremental data transfer described in Algorithm 3 has a lower time complexity, and this also means that it has a lower runtime scheduling overhead.

C. COMMUNICATION OPTIMIZATION BASED ON SOFTWARE PIPELINING

Another effective way to reduce inter-device communication cost is to overlap data transfers with kernel execution, this subsection presents a three-way overlapping communication optimization method based on software pipelining.

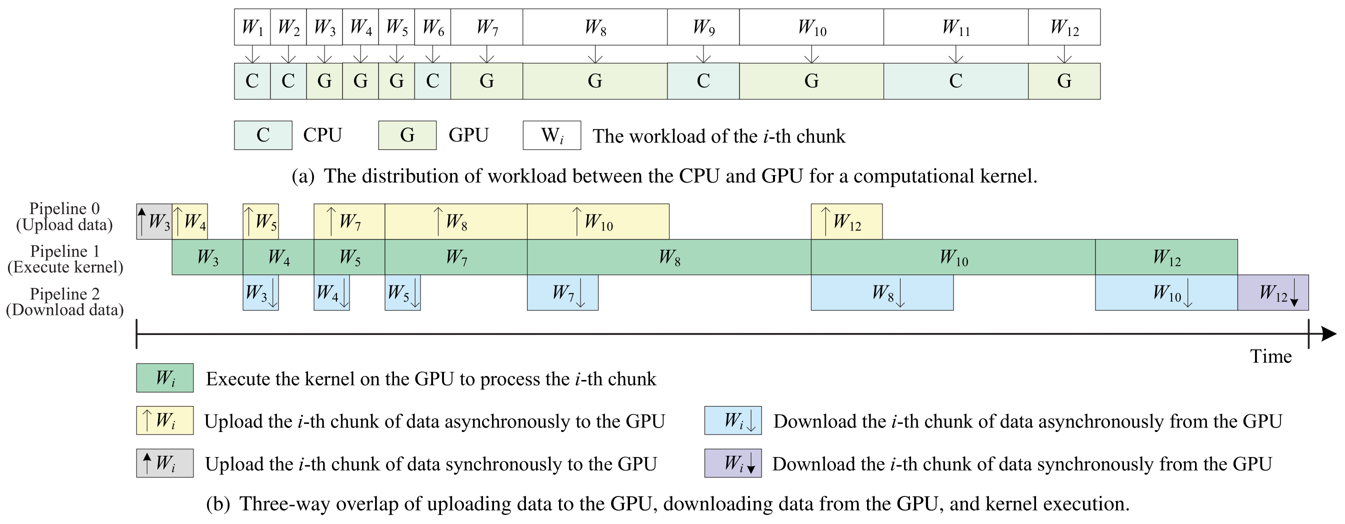
1) THE THREE-WAY OVERLAPPING COMMUNICATION OPTIMIZATION METHOD BASED ON SOFTWARE PIPELINING

The three-way overlapping communication optimization method relies on two things: (i) the "chunked" computation, i.e., the entire iteration space of a data-parallel for-loop is split into several chunks, and multiple devices are used to cooperatively process these chunks; (ii) the "three-way" overlap of uploading data to the accelerator, downloading data from the accelerator, and kernel execution.

In this work, we use three software pipelines that can be run in parallel to achieve the overlap of data transfers and kernel execution. Specifically, the first pipeline is responsible for asynchronously uploading the next chunk of data to the accelerator, the second pipeline is responsible for asynchronously processing the current chunk on the accelerator, and the third pipeline is responsible for asynchronously downloading the previous chunk of data from the accelerator.

To better understand the inter-device communication optimization method described above, an example of the CPU-GPU communication optimization based on software pipelining is illustrated in Fig. 8. For simplicity, assuming that the data transfers and kernel execution take roughly the same amount of time without considering the communication optimization, namely the data upload, kernel execution, and data download take up around 25%, 50%, and 25% of the total running time, respectively.

Fig. 8(a) shows the distribution of workload between the CPU and GPU for a computational kernel. Fig. 8(b) shows the three-way overlap of uploading data to the GPU,



W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 W11 W12
C C G G G C G G C G C G
C CPU
G GPU
Wi The workload of the i-th chunk
(a) The distribution of workload between the CPU and GPU for a computational kernel.
Pipeline 0 (Upload data)
Pipeline 1 (Execute kernel)
Pipeline 2 (Download data)
Time
Execute the kernel on the GPU to process the i-th chunk
Upload the i-th chunk of data asynchronously to the GPU
Upload the i-th chunk of data synchronously to the GPU
Download the i-th chunk of data asynchronously from the GPU
Download the i-th chunk of data synchronously from the GPU
(b) Three-way overlap of uploading data to the GPU, downloading data from the GPU, and kernel execution.

FIGURE 8. **An example of the CPU-GPU communication optimization based on software pipelining.**

downloading data from the GPU, and kernel execution. Specifically, firstly, the first chunk (i.e., W_3) of data need to be synchronously uploaded to the GPU before processing the first chunk on the GPU. Secondly, pipeline 0 asynchronously uploads the second chunk (i.e., W_4) of data to the GPU while pipeline 1 begins asynchronously processing the first chunk. Thirdly, pipeline 0 asynchronously uploads the third chunk (i.e., W_5) of data to the GPU, pipeline 1 asynchronously processes the second chunk on the GPU, and pipeline 2 asynchronously downloads the first chunk of data from the GPU. Fourthly, except for the processing of the last chunk (i.e., W_{12}), we repeat the following operations: pipeline 0 uploads the next chunk of data and pipeline 2 downloads the previous chunk of data while pipeline 1 is processing the current chunk. Finally, pipeline 2 asynchronously downloads the second-to-last chunk (i.e., W_{10}) of data from the GPU while pipeline 1 begins asynchronously processing the last chunk, and the last chunk of data need to be synchronously downloaded from the GPU after the last chunk has finished processing.

In the example illustrated in Fig. 8, it is readily seen that the software pipelining mechanism can hide all of the data transfers between the CPU and GPU, except for uploading the first chunk of data and downloading the last chunk of data.

2) THE MODIFIED ADETS THAT SUPPORTS THREE-WAY OVERLAPPING COMMUNICATION OPTIMIZATION

Considering that the next chunk of data need to be uploaded to the accelerator while the current chunk is being processed on the accelerator, how to determine the appropriate size of the next chunk before processing the current chunk is a key problem. In order to solve this problem and to support the overlap of data transfers and kernel execution, we make some modifications to ADETS. The modified ADETS that supports three-way overlapping communication optimization based on software pipelining is described in Algorithm 4. Each execution of the computational kernel consists of the following steps.

Step 1: We firstly assign a chunk whose size is $W_{\text{curr}.i}$ to device D_i and update the assigned workload W_a, where $W_a = W_a + W_{\text{curr}.i}$ and $1 \le i \le p$. If this is the first execution of the kernel, $W_{\text{curr}.i} = W_{\text{next}.i} = W/n$; otherwise, we find a chunk $W_{\text{fs}.i}$ processed by device D_i at the fastest speed from the previous execution of the kernel, and $W_{\text{curr}.i} = W_{\text{next}.i} = W_{\text{fs}.i}$. Secondly, we pre-assign a subsequent chunk whose size is $W_{\text{next}.i}$ to device D_i and update the assigned workload W_a and unassigned workload W_u, where $W_a = W_a + W_{\text{next}.i}$ and $W_u = W - W_a$. Thirdly, we execute the kernel on device D_i to process the first chunk assigned to it. If device D_i is an accelerator, we need to synchronously upload the first chunk of data to D_i before processing the first chunk assigned to it, and then we need to asynchronously upload the second chunk of data to device D_i while processing the first chunk. After device D_i has completed its work, we obtain the execution time of device D_i to compute its execution speed.

Step 2: We firstly pre-assign the next chunk to device D_i if there is unassigned workload and update the assigned and unassigned workloads. Secondly, we use device D_i to process the second chunk assigned to it. If device D_i is an accelerator, we use pipeline 0 to asynchronously upload the next chunk of data to device D_i, we use pipeline 1 to asynchronously process the current chunk on device D_i, and we use pipeline 2 to asynchronously download the previous chunk of data from device D_i. After device D_i has completed its work, we compute the current execution speed $V_{\text{curr}.i}$ of device D_i. Thirdly, we determine the size of the chunk after the next chunk that will be assigned to device D_i (i.e., $W_{\text{next\_next}.i}$) by comparing $V_{\text{prev}.i}$ and $V_{\text{curr}.i}$. If $|V_{\text{curr}.i} - V_{\text{prev}.i}| \le V_{\text{prev}.i} \times 10\%$, $W_{\text{next\_next}.i} = (W_{\text{curr}.i} + W_{\text{next}.i})/2$; otherwise, $W_{\text{next\_next}.i} = W_{\text{curr}.i} + W_{\text{next}.i}$ when $V_{\text{curr}.i} > V_{\text{prev}.i}$, while $W_{\text{next\_next}.i} = (W_{\text{curr}.i} + W_{\text{next}.i})/4$ when $V_{\text{curr}.i} < V_{\text{prev}.i}$. Finally, we update the previous execution speed $V_{\text{prev}.i}$ of device D_i.

Algorithm 4 The Modified ADETS That Supports Three-Way Overlapping Communication Optimization

```
Require: $p$, $W$, the initial chunk size $W/n$, and $tolExecs$
1:  for $t = 1$ to $tolExecs$ do
2:    Initialize $W_{prev.i} = 0$, $W_a = 0$, $W_u = W$, and $V_{prev.i} = 0$;
3:    for each controller thread $t_i$, $1 \le i \le p$, in parallel do
4:      if $t == 1$ then
5:        $W_{curr.i} = W_{next.i} = W_{next\_next.i} = W/n$;
6:      else
7:        Find a chunk $W_{fs.i}$ processed by $D_i$ at the fastest speed
          from the $(t-1)$-th execution of the kernel;
8:        $W_{curr.i} = W_{next.i} = W_{next\_next.i} = W_{fs.i}$;
9:      end if
10:     Assign a chunk whose size is $W_{curr.i}$ to $D_i$;
11:     Update the assigned workload: $W_a = W_a + W_{curr.i}$;
12:     while $W_u > 0$ or $W_{next.i} > 0$ do
13:       if $W_{next.i} > 0$ then
14:         Pre-assign a chunk whose size is $W_{next.i}$ to $D_i$;
15:       end if
16:       Update the assigned workload: $W_a = W_a + W_{next.i}$;
17:       Update the unassigned workload: $W_u = W - W_a$;
18:       if device $D_i$ is an accelerator then
19:         if $W_{prev.i} == 0$ then
20:           Upload the first chunk of data to $D_i$;
21:         end if
22:         if $W_{next.i} > 0$ then
23:           Use pipeline 0 to asynchronously upload the next
              chunk of data to $D_i$;
24:         end if
25:         Use pipeline 1 to asynchronously execute the kernel
            on $D_i$ to complete the workload $W_{curr.i}$;
26:         if $W_{prev.i} > 0$ then
27:           Use pipeline 2 to asynchronously download the
              previous chunk of data from $D_i$;
28:         end if
29:         if $W_{next.i} == 0$ then
30:           Download the last chunk of data from $D_i$;
31:         end if
32:       else
33:         Execute the kernel on $D_i$ to complete $W_{curr.i}$;
34:       end if
35:       Obtain the current execution time $T_{curr.i}$ of $D_i$;
36:       Compute the current execution speed of $D_i$:
          $V_{curr.i} = W_{curr.i}/T_{curr.i}$;
37:       if $V_{prev.i} > 0$ then
38:         Determine the size of the chunk after the next chunk
            assigned to $D_i$ (i.e., $W_{next\_next.i}$) by comparing $V_{prev.i}$
            and $V_{curr.i}$;
39:       end if
40:       if $W_{next\_next.i} > W_u$ then $W_{next\_next.i} = W_u$; end if
41:       $W_{prev.i} = W_{curr.i}$, $W_{curr.i} = W_{next.i}$, $W_{next.i} =$
    $W_{next\_next.i}$;
42:       Update the previous execution speed: $V_{prev.i} = V_{curr.i}$;
43:     end while
44:   end for
45: end for
```

Step 3: Repeat Step 2 until the unassigned workload has finished assignment and all the chunks assigned to device D_i have finished processing. If device D_i is an accelerator, we need to synchronously download the last chunk of data from D_i after the last chunk assigned to it has finished processing.

Compared with the original ADETS, the modified ADETS with communication optimization does not address the load imbalance that might happen at the end of the entire iteration space. Although this may result in some performance loss, the overall performance can still be greatly improved due to a significant reduction in the inter-device communication overhead.

When a heterogeneous system has p compute devices that participate in multi-device co-execution and a computational kernel needs to be executed λ times repeatedly, the time complexity of the modified ADETS that supports three-way overlapping communication optimization described in Algorithm 4 is $O(\lambda W/p)$ in the worst case, where W is the total workload of the computational kernel. It is readily seen that the modified ADETS that supports three-way overlapping communication optimization can provide a low-overhead runtime scheduling.

TABLE 1. A subset of the runtime APIs provided by HCE.

Category	Function Prototype
task scheduling	void hce_co_schedule(string schedType, int initialChunkSize) void hce_set_ratio(string deviceType, int deviceNum, float partitionRatio)
device management	void hce_init_compute_device(int deviceUDI, string deviceType, int deviceNum) void hce_get_device_udi(string deviceType, int deviceNum) void hce_get_num_devices(string deviceType)
memory management	void hce_malloc_gpu(int deviceNum, void** devPtr, int length) void hce_free_gpu(int deviceNum, void* devPtr) void hce_malloc_mic(int deviceNum, T *hostPtr, int length) void hce_free_mic(int deviceNum, T *hostPtr) void hce_memcpy_to_gpu(int deviceNum, void *devPtr, void *hostPtr, int begin, int end, int stride) void hce_memcpy_from_gpu(int deviceNum, void *hostPtr, void *devPtr, int begin, int end, int stride) void hce_memcpy_to_gpu_async(int deviceNum, void *devPtr, void *hostPtr, int begin, int end, int stride, cudaStream_t stream) void hce_memcpy_from_gpu_async(int deviceNum, void *hostPtr, void *devPtr, int begin, int end, int stride, cudaStream_t stream) void hce_memcpy_to_mic(int deviceNum, T *hostPtr, int begin, int end, int stride) void hce_memcpy_from_mic(int deviceNum, T *hostPtr, int begin, int end, int stride)
transfer optimization	struct requiredArraySection *hce_get_array_section_upload(int numOfDevices, int deviceUDI, int loopBegin, int loopEnd, float *previousPartitionRatios, float *currentPartitionRatios) struct requiredArraySection *hce_get_array_section_download(int numOfDevices, int deviceUDI, int loopBegin, int loopEnd, float *currentPartitionRatios, float *nextPartitionRatios)

V. THE IMPLEMENTATION OF HCE

This section presents the runtime APIs provided by HCE and an example of using HCE.

A. THE RUNTIME APIs PROVIDED BY HCE

As shown in Fig. 1, our proposed HCE provides a set of runtime APIs for task scheduling, device management, memory management, and transfer optimization. A subset of the runtime APIs provided by HCE are listed in Table 1.

In our proposed HCE, task scheduling aims to effectively split work across devices, such as FDETS, ADETS, the modified FDETS that supports incremental data transfer, and the modified ADETS that supports three-way overlapping communication optimization. Device management is responsible for keeping and updating the required information for each compute device that participates in multi-device co-execution, such as the begin and end positions of the iteration space assigned to the specified compute device. Memory management is responsible for the allocation and deallocation of accelerator memory and the data transfer between devices. Transfer optimization aims to effectively reduce inter-device communication overhead, such as our proposed incremental data transfer method and three-way overlapping communication optimization method based on software pipelining.

B. EXAMPLE OF USING HCE

Fig. 9 shows an example of using the hybrid OpenMP/CUDA parallel programming model and the runtime APIs provided by HCE to write a matrix addition program that can be cooperatively executed on a hybrid CPU-GPU system. The implementation of the matrix addition consists of the following seven key steps.

Step 1: Programmers need to set the number of compute devices that participate in multi-device co-execution (see line 2 in Fig. 9).

Step 2: Programmers need to specify the begin and end positions of the outermost for-loop of the computational kernel (see line 4).

Step 3: Programmers can use the hybrid OpenMP/CUDA parallel programming model to write two device-specific computational kernels and a CUDA kernel function (see lines 6-20). Specifically, the CPU kernel mainly contains an OpenMP parallel region (see lines 6-10) and the GPU kernel mainly includes data transfer and CUDA kernel launch codes (see lines 15-20).

Step 4: Programmers can use some runtime APIs related to memory management to handle the accelerator memory allocation and deallocation (see lines 24-26 and 34-36).

Step 5: Programmers can use some runtime APIs related to device management to specify the unique device ID, device type, device number, and the computational kernel function of each device (see lines 27-30).

Step 6: Programmers need to specify the initial partition ratios that will be used in the task scheduling (see lines 31-32).

Step 7: Programmers need to specify the task scheduling scheme and the initial chunk size and to launch the task scheduling (see line 33).

As noted above, with the aid of HCE, it is not necessary for programmers to know which chunks of the computation are going to be scheduled to which device or to specify which parts of the data are going to be copied to/from which accelerator, because these complicated and cumbersome works are automatically performed by HCE.

```
 1 struct computeDevice {int deviceUDI; string deviceType; int deviceNum; float ratio;
     void (*comp_kernel)(int deviceNum, int subLoopBegin, int subLoopEnd); };
 2 int numOfDevices = 2;
 3 struct computeDevice devices[numOfDevices];
 4 int M = 8192, N = 8192, loopBegin = 0, loopEnd = M−1;
 5 double *A, *B, *C, *devA, *devB, *devC;
 6 void CPU_kernel(int deviceNum, int subLoopBegin, int subLoopEnd) {
 7   #pragma omp parallel for num_threads(omp_get_num_procs()−numOfDevices+1)
 8   for(int i = subLoopBegin; i <= subLoopEnd; i++)
 9      for(int j = 0; j < N; j++)
10         C[i*N+j] = A[i*N+j] + B[i*N+j]; }
11 __global__ void CUDA_kernel(int subLoopBegin, int subLoopEnd, int N,
     double *A, double *B, double *C) {
12   for(int i = subLoopBegin + blockIdx.x*1; i <= subLoopEnd; i += gridDim.x*1)
13      for(int j = 0 + threadIdx.x*1; j < N; j += blockDim.x*1)
14         C[i*N+j] = A[i*N+j] + B[i*N+j]; }
15 void GPU_kernel(int deviceNum, int subLoopBegin, int subLoopEnd) {
16   if(initial_run == true) hce_memcpy_to_gpu(deviceNum, devB, B, 0, M*N);
17   hce_memcpy_to_gpu(deviceNum, devA, A, subLoopBegin, subLoopEnd, N);
18   cudaSetDevice(deviceNum);
19   CUDA_kernel<<<1024, 256>>>(subLoopBegin, subLoopEnd, N, devA, devB,
        devC);
20   hce_memcpy_from_gpu(deviceNum, C, devC, subLoopBegin, subLoopEnd, N);}
21 void main() {
22   //allocate host memory for A[M*N], B[M*N], and C[M*N].
23   //initialize A and B.
24   hce_malloc_gpu(0, (void**)&devA, M*N);
25   hce_malloc_gpu(0, (void**)&devB, M*N);
26   hce_malloc_gpu(0, (void**)&devC, M*N);
27   hce_init_compute_device(0, “CPU”, 0);
28   hce_init_compute_device(1, “GPU”, 0);
29   devices[0].comp_kernel = CPU_kernel;
30   devices[1].comp_kernel = GPU_kernel;
31   hce_set_ratio(“CPU”, 0, 0.35);
32   hce_set_ratio(“GPU”, 0, 0.65);
33   hce_co_schedule(“FDETS”, M / 128);
34   hce_free_gpu(0, devA);
35   hce_free_gpu(0, devB);
36   hce_free_gpu(0, devC);
37   //deallocate host memory for A, B, and C. }
```

FIGURE 9. The matrix addition written using the hybrid OpenMP/CUDA parallel programming model and the runtime APIs provided by HCE.

VI. EXPERIMENTAL EVALUATION

This section first presents the experimental setup, next evaluates the effectiveness of our proposed inter-device communication optimization methods, then evaluates the performance of HCE's multi-device co-execution, and finally presents the performance comparison among HCE, StarPU, and OmpSs.

A. EXPERIMENTAL SETUP

A serials of experiments are conducted on the following two different test platforms: (i) a hybrid CPU-GPU-MIC system consisting of two Intel Xeon E5-2640v2 CPUs, an NVIDIA Tesla K40c GPU, an Intel Xeon Phi 7110P Coprocessor, and 64GB host memory; (ii) a hybrid CPU-GPU-GPU system consisting of two Intel Xeon E5-2680v4 CPUs, two NVIDIA Tesla P100 GPUs, and 256GB host memory. On each test platform, the multi-core CPUs and all many-core accelerators are interconnected through the PCI-E bus. The specifications of these CPUs and accelerators are as follows:

- Xeon E5-2640v2 CPU: 8 cores at 2.0GHz.
- Xeon E5-2680v4 CPU: 14 cores at 2.4GHz.
- Tesla K40c GPU: 2880 CUDA cores at 745MHz, 12GB device memory, and 288GB/s memory bandwidth.
- Tesla P100 GPU: 3584 CUDA cores at 1189MHz, 16GB device memory, and 732GB/s memory bandwidth.
- Xeon Phi 7110P Coprocessor: 61 cores at 1.1GHz, 8GB device memory, and 352GB/s memory bandwidth.

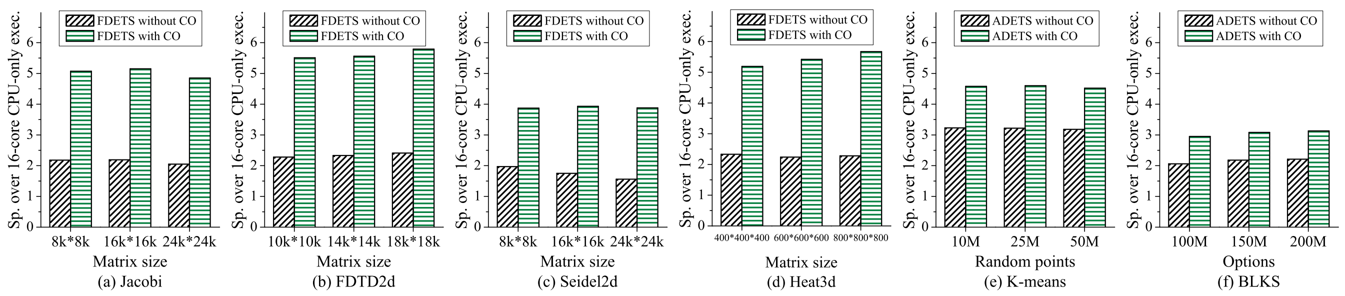


FIGURE 10. Comparison of the performance before and after the inter-device communication optimization implemented in the CPU-GPU-MIC co-execution.

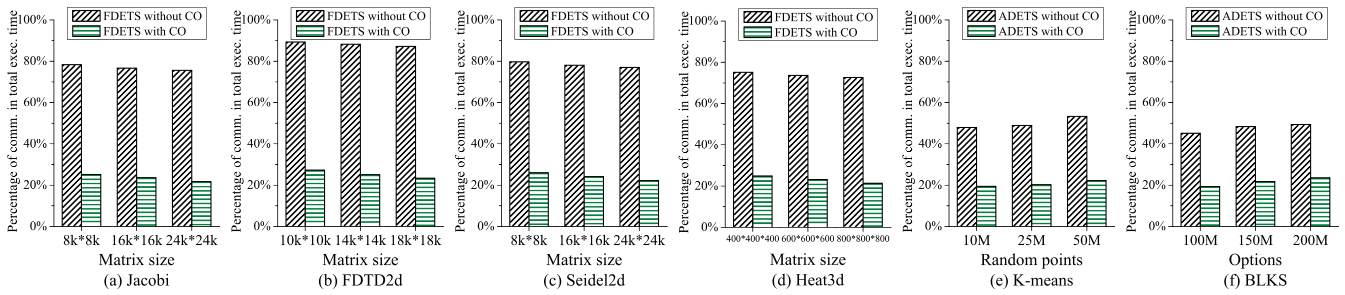


FIGURE 11. The percentage of time spent on inter-device communication in the CPU-GPU-MIC co-execution.

In software, these two test platforms are built on top of the Red Hat Enterprise Linux Server release 8.0 operating system with NVIDIA CUDA Toolkit 10.2 and Intel C++ Compiler 17.0 using auto-vectorization and optimization flag *-Ofast*.

Table 2 lists 8 representative benchmarks: Jacobi, FDTD2d, Seidel2d, and Heat3d are from the PolyBench suite [29]; K-means and BFS are from the Rodinia benchmark suite [30]; GEMM and BLKS are from the NVIDIA CUDA SDK [31]. Each benchmark includes one or more data-parallel kernels, and OpenMP, CUDA, and Intel Offload are responsible for the parallelization of the outermost for-loop within each data-parallel kernel on the CPU, GPU, and MIC, respectively. In our experiments, for each benchmark with different problem sizes, we use a random number generator to produce 100 different instances, and the average execution time of 100 different instances is considered.

B. EVALUATION OF THE INTER-DEVICE COMMUNICATION OPTIMIZATION METHODS

To evaluate the effectiveness of our proposed communication optimization methods, we run Jacobi, FDTD2d, Seidel2d, Heat3d, K-means, and BLKS on the hybrid CPU-GPU-MIC system using FDETS without communication optimization (FDETS without CO) and ADETS without communication optimization (ADETS without CO) proposed in [23], FDETS with communication optimization (FDETS with CO) described in Section IV-B2, and ADETS with communication optimization (ADETS with CO) described in Section IV-C2. The incremental data transfer method is used in the CPU-GPU-MIC co-execution of Jacobi, FDTD2d, Seidel2d, and Heat3d. The three-way overlapping communication optimization method based on software pipelining is used in the CPU-GPU-MIC co-execution of K-means and BLKS.

Fig. 10 shows a comparison of the performance before and after the inter-device communication optimization implemented in the CPU-GPU-MIC co-execution of six different benchmarks. The results show that the two communication optimization methods significantly improve the overall performance of multi-device co-execution. Compared with FDETS without CO, FDETS with CO achieves an average speedup of 2.35×, 2.40×, 2.23×, and 2.38× for Jacobi, FDTD2d, Seidel2d, and Heat3d, respectively. Similarly, compared with ADETS without CO, ADETS with CO achieves an average improvement in performance of 42.17% and 41.95% for K-means and BLKS, respectively.

To clearly explain the reasons for the performance improvement, we measure the inter-device communication time and the total execution time of CPU-GPU-MIC co-execution. Fig. 11 presents the percentage of time spent on inter-device communication in the CPU-GPU-MIC co-execution of six different benchmarks. When using FDETS without CO, the inter-device communication takes up about 77%, 88%, 78%, and 74% of the total execution

TABLE 2. Benchmarks used in our experiments.

Bench-mark	Input Problem Size		
	Small	Medium	Large
Jacobi	$8k \times 8k$ (matrix size)	$16k \times 16k$	$24k \times 24k$
FDTD2d	$10k \times 10k$ (matrix size)	$14k \times 14k$	$18k \times 18k$
Seidel2d	$8k \times 8k$ (matrix size)	$16k \times 16k$	$24k \times 24k$
Heat3d	$400 \times 400 \times 400$ (matrix size)	$600 \times 600 \times 600$	$800 \times 800 \times 800$
K-means	$10M$ (random points)	$25M$	$50M$
BFS	$4M$ (nodes)	$8M$	$16M$
GEMM	$4k \times 4k$ (matrix size)	$8k \times 8k$	$12k \times 12k$
BLKS	$100M$ (options)	$150M$	$200M$

time of Jacobi, FDTD2d, Seidel2d, and Heat3d, respectively. However, when using FDETS with CO, it only takes up about 23%, 25%, 24%, and 23% of the total execution time of Jacobi, FDTD2d, Seidel2d, and Heat3d, respectively. Similarly, the inter-device communication times of ADETS with CO are reduced by an average of 70.99% and 68.11% than that of ADETS without CO for K-means and BLKS, respectively. The results show that the communication overhead between devices can be greatly reduced with the help of our proposed two communication optimization methods. Specifically, a large amount of redundant data transfers between host and accelerator are avoided by using the incremental data transfer method for Jacobi, FDTD2d, Seidel2d, and Heat3d, and a great deal of data transfer overhead between host and accelerator could be hidden by using the three-way overlapping communication optimization method for K-means and BLKS.

C. EVALUATION OF HCE's MULTI-DEVICE CO-EXECUTION

In this subsection, we first compare the performance of HCE's multi-device co-execution with that of the best single-device execution on the hybrid CPU-GPU-MIC system, and then evaluate the performance of HCE's CPU-GPU-GPU co-execution.

1) COMPARISON WITH THE BEST SINGLE-DEVICE EXECUTION

This subsection compares the performance of HCE's CPU-GPU-MIC co-execution with that of the best single-device execution. In this experiment, the best single-device execution refers to the best one among the 16-core CPU-only, GPU-only and MIC-only executions. To better evaluate the performance of HCE's multi-device co-execution, a suitable inter-device task scheduling scheme is adopted for each benchmark. Specifically, FDETS without CO is adopted for GEMM; ADETS without CO is adopted for BFS; FDETS with CO is adopted for Jacobi, FDTD2d, Seidel2d, and Heat3d; ADETS with CO is adopted for K-means and BLKS.

Fig. 12 demonstrates the speedup of HCE's CPU-GPU-MIC co-execution over the best single-device execution for each benchmark with small, medium, and large problem sizes. The results show that HCE's CPU-GPU-MIC co-execution is much faster than the best single-device execution in most cases, e.g., it achieves performance improvements of up to 1.14×, 1.16×, and 1.43× in Seidel2d, K-means, and BLKS, respectively. However, the modest performance improvement of 0.32× is observed in BFS. This is because BFS runs much faster on the best compute device as compared to other compute devices. The best compute device undertakes the majority of the work in the CPU-GPU-MIC co-execution of BFS. As a consequence, the other compute devices contribute less to the overall performance of CPU-GPU-MIC co-execution.

In general, a proper inter-device task scheduling scheme should be adopted according to different data-parallel applications. FDETS without CO and ADETS without CO are

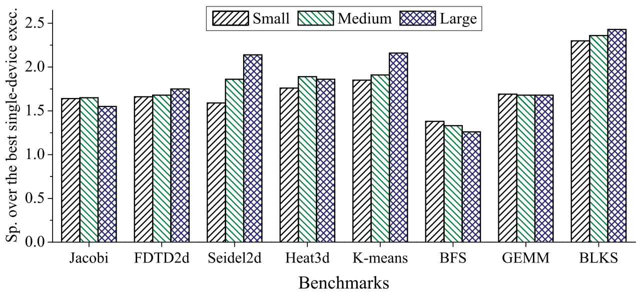


FIGURE 12. The speedups of HCE's CPU-GPU-MIC co-execution over the best single-device execution for different benchmarks.

applicable to some data-parallel applications that have small inter-device communication overhead, FDETS with CO is suitable for some data-parallel applications which have one or more computational kernels that need to be executed repeatedly and may contain a large amount of redundant data transfers, and ADETS with CO is suitable for some data-parallel applications whose multi-device co-execution may incur huge inter-device communication overhead.

2) PERFORMANCE EVALUATION ON MULTIPLE GPUs

This subsection evaluates the performance of HCE's CPU-GPU-GPU co-execution. In this experiment, a suitable inter-device task scheduling scheme is adopted for each benchmark as described in Section VI-C1.

Fig. 13 shows a performance comparison of the 28-core CPU-only execution, CPU-GPU co-execution, and CPU-GPU-GPU co-execution for different benchmarks with large problem size. It is obvious that HCE's multi-device co-execution is much faster than the CPU-only execution. Specifically, compared with the 28-core CPU-only execution, CPU-GPU-GPU co-execution achieves speedups of up to 6.91×, 6.33×, 5.41×, 9.96×, 12.70×, 6.07×, 8.23×, and 5.06× in Jacobi, FDTD2d, Seidel2d, Heat3d, K-means, BFS, GEMM, and BLKS, respectively. The performance benefit mainly comes from the full utilization of two Intel Xeon 14-core E5-2680v4 CPUs and two Tesla P100 GPUs each with 3584 CUDA cores of the hybrid CPU-GPU-GPU system. We believe that this is also due to the good load balancing and lower communication overhead between devices.

Fig. 13 also shows that the performance is improved with an increase in GPUs. Compared with the CPU-GPU co-execution, CPU-GPU-GPU co-execution achieves a 77.96%, 77.03%, 72.71%, 76.23%, 85.55%, 74.11%, 80.18%, and 71.69% performance improvements for Jacobi, FDTD2d, Seidel2d, Heat3d, K-means, BFS, GEMM, and BLKS, respectively. The results show that an additional GPU brings significant performance improvement for most benchmarks, as the GPU performs much better than the CPU and the increased CPU-GPU communication cost is small.

Although the two 14-core E5-2680v4 CPUs perform worse than the Tesla P100 GPU for most benchmarks, HCE assigns a small amount of work to the two CPUs and achieves a good load balance. Fig. 14 shows the execution time comparison

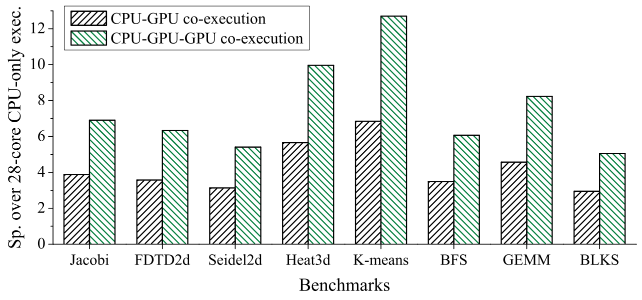


FIGURE 13. Performance evaluation on multiple GPUs.

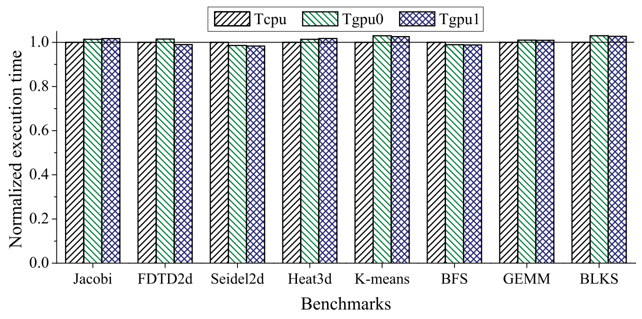


FIGURE 14. Execution time comparison among CPU, GPU 0, and GPU 1 in the CPU-GPU-GPU co-execution of each benchmark.

among CPU, GPU 0, and GPU 1 in the CPU-GPU-GPU co-execution of each benchmark. In Fig. 14, T_{cpu}, T_{gpu0}, and T_{gpu1} denote the time the CPU, GPU 0, and GPU 1 take to complete its assigned workload, respectively. From the figure, we see that the difference in the execution time of each compute device is very small for most benchmarks. The good load balance between devices mainly benefits from our proposed inter-device task scheduling schemes.

D. COMPARISON WITH StarPU AND OmpSs

This subsection compares the performance of HCE with that of StarPU [14] and OmpSs [15]. StarPU is a runtime system that offers a unified view of the computational resources to allow programmers to exploit the computing power of the available CPUs and accelerators, while transparently handling low-level issues such as data transfers in a portable fashion. It provides task programming APIs for data partitioning and task scheduling across heterogeneous devices. OmpSs provides a task-based programming model where users can offload computation and data to multiple devices by adding OmpSs directives and clauses, and it is able to schedule tasks in a data flow way to the available CPUs and accelerators based on the task graph built at runtime.

In this experiment, we implement 8 benchmarks using our proposed HCE, the newest StarPU 1.3.8's rich C APIs [32], and OmpSs 19.06's directives [33] on the hybrid CPU-GPU-GPU system. The performance model-based *dmda* (deque model data aware) scheduler is used in StarPU, which schedules tasks where their termination time will be minimal with taking task execution performance models and data transfer time into account. The versioning scheduler is used in OmpSs, which can automatically profile each task implementation and choose the most suitable implementation each time the task must be run. Given that StarPU and OmpSs support the overlap of data transfers with computation, our proposed FDETS with communication optimization and ADETS with communication optimization are adopted in HCE.

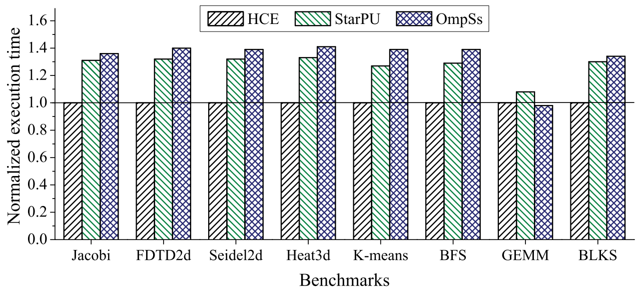


FIGURE 15. Performance comparison among HCE's, StarPU's, and OmpSs's GPU-GPU co-execution for different benchmarks.

Fig. 15 gives the performance comparison among HCE's, StarPU's, and OmpSs's GPU-GPU co-execution for different benchmarks with large problem size. The results show that HCE yields better performance than StarPU and OmpSs for some data-parallel applications. For example, compared with StarPU, HCE achieves 31.07%, 32.17%, and 29.32% performance improvements for Jacobi, FDTD2d, and BFS, respectively; compared with OmpSs, HCE achieves 38.53%, 39.39%, and 33.72% performance improvements for Seidel2d, K-means, and BLKS, respectively. Note that OmpSs achieves a slightly better performance than HCE for some data-parallel kernels (such as GEMM) that need to be executed one time and have a small inter-device communication overhead. For these 8 benchmarks, HCE achieves an average of 27.61% and 33.17% performance improvements over StarPU and OmpSs, respectively. The performance improvement is mainly due to HCE's inter-device task scheduling schemes can provide lower runtime scheduling overhead and higher device utilization and effectively reduce the data transfer overhead between devices. Although our proposed HCE performs better than StarPU and OmpSs for some data-parallel applications, this does not mean that HCE can replace StarPU and OmpSs, because they have their own advantages, disadvantages, and limitations.

VII. RELATED WORK

Heterogeneous CPU-accelerator systems have come into common use recently. Some directive-based parallel programming models have been developed as a powerful way to easily harness the computing power of many-core accelerators, such as hiCUDA [34], OpenMPC [35], and OpenACC [36]. They allow programmers to use directives to identify which parts of a program should be automatically offloaded to an accelerator, but they do not allow for offloading parallel codes to multiple CPUs and accelerators. Unlike these works, Shuja *et al.* [37] proposed a framework for single instruction multiple data instruction translation and offloading for mobile devices (SIMDOM) in heterogeneous mobile and cloud environments, which allows mobile applications to be executed on edge and cloud servers, and various modules of the SIMDOM framework for optimal execution parameters are analyzed systematically and comprehensively in [38].

Some heterogeneous parallel programming models and runtime systems [9]–[15] have recently focused on how to fully utilize multiple compute devices to execute parallel applications on a heterogeneous system. To make full use of multiple compute devices in OpenCL, SKMD [9] provides an OpenCL runtime for heterogeneous devices, which takes a kernel written for a single device and executes it across multiple devices. Similar to SKMD, CoopCL [10] also provides an OpenCL runtime that targets CPU-GPU systems, which takes applications written for a single device and automatically runs each kernel on both CPU and GPU. EngineCL [11] presents an OpenCL-based runtime system that effectively splits the workload of a single massive data-parallel kernel to multiple different compute devices so as to maximize their utilization. FinePar [12] offers a software framework that enables the fine-grained workload partitioning between the CPU and GPU on the same die for irregular applications written in OpenCL. Most of existing parallel applications are written in OpenMP, if SKMD, CoopCL, EngineCL, or FinePar is adopted to implement the multi-device co-execution of these applications, programmers need to make a big effort to rewrite these applications using OpenCL.

CoreTSAR [13] can automatically schedule data-parallelism tasks between CPU and GPU based on Accelerated OpenMP. It supports co-scheduling of parallel loop regions across an arbitrary number of CPUs and GPUs. In our previous work [23], we have discussed CoreTSAR's two dynamic scheduling strategies: quick scheduling and split scheduling. Both StarPU [14] and OmpSs [15] are most closely related to our proposed HCE. In Section VI-D, StarPU and OmpSs are introduced in detail, and the performance comparison among StarPU, OmpSs, and HCE are made. As shown in Fig. 15, the results show that HCE can achieve better performance than StarPU and OmpSs for some data-parallel applications. Moreover, HCE supports the more efficient data transfer between devices in comparison with StarPU and OmpSs.

In a nutshell, our HCE provides efficient inter-device task scheduling strategies and inter-device communication optimization methods and some easy-to-use runtime APIs, which can help programmers to automatically and efficiently map computation and data to multiple compute devices on a heterogeneous CPU-accelerator system.

VIII. CONCLUSION

In this paper, we present HCE, a runtime system that efficiently supports the heterogeneous cooperative execution of data-parallel applications on hybrid CPU-accelerator systems. HCE provides a simple and effective way for application programmers to fully exploit the available compute devices to accelerate their applications, reducing the burden on programmers and allowing them to concentrate their attention on the application itself. In order to effectively reduce the communication overhead between devices, we propose two inter-device communication optimization methods, and which have been integrated into the inter-device task scheduling schemes. A prototype of HCE is built on hybrid CPU-accelerator systems. The experimental results show that the data transfer overhead can be greatly reduced with the help of our proposed inter-device communication optimization methods and the multi-device co-execution using HCE provides much better performance than the best single-device execution. Compared with the widely used StarPU and OmpSs, HCE also achieves a better performance for some data-parallel applications.

In future work, we plan to extend HCE to support efficient execution of data-parallel kernels on heterogeneous CPU-accelerator clusters. Moreover, considering that the thread configuration would affect the performance of a compute device, we will explore how to dynamically determine the best thread configuration of each compute device according to the workload assigned to it during runtime.

REFERENCES

[1] J. Kim and B. Nam, "Co-processing heterogeneous parallel index for multi-dimensional datasets," *J. Parallel Distrib. Comput.*, vol. 113, pp. 195–203, Mar. 2018.

[2] M. Tayyub and G. N. Khan, "Heterogeneous design and efficient CPU-GPU implementation of collision detection," *IADIS Int. J. Comput. Sci. Inf. Syst.*, vol. 14, no. 2, pp. 25–40, Dec. 2019.

[3] H. Zou, S. Tang, C. Yu, H. Fu, Y. Li, and W. Tang, "ASW: Accelerating Smith–Waterman algorithm on coupled CPU–GPU architecture," *Int. J. Parallel Program.*, vol. 47, no. 3, pp. 388–402, Jun. 2019.

[4] O. Pearce, "Exploring utilization options of heterogeneous architectures for multi-physics simulations," *Parallel Comput.*, vol. 87, pp. 35–45, Sep. 2019.

[5] T. S. Abdelrahman, "Cooperative software-hardware acceleration of K-means on a tightly coupled CPU-FPGA system," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 3, pp. 1–24, Aug. 2020.

[6] L. Mabrouk, S. Huet, D. Houzet, S. Belkouch, A. Hamzaoui, and Y. Zennayi, "Efficient adaptive load balancing approach for compressive background subtraction algorithm on heterogeneous CPU–GPU platforms," *J. Real-Time Image Process.*, vol. 17, no. 5, pp. 1567–1583, Oct. 2020.

[7] N. Naz, A. Haseeb Malik, A. B. Khurshid, F. Aziz, B. Alouffi, M. I. Uddin, and A. AlGhamdi, "Efficient processing of image processing applications on CPU/GPU," *Math. Probl. Eng.*, vol. 2020, Oct. 2020, Art. no. 4839876.

[8] R. Souza, A. Fernandes, T. S. F. X. Teixeira, G. Teodoro, and R. Ferreira, "Online multimedia retrieval on CPU–GPU platforms with adaptive work partition," *J. Parallel Distrib. Comput.*, vol. 148, pp. 31–45, Feb. 2021.

[9] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "SKMD: Single kernel on multiple devices for transparent CPU-GPU collaboration," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 1–27, Sep. 2015.

[10] K. Moren and D. Gohringer, "CoopCL: Cooperative execution of OpenCL programs on heterogeneous CPU-GPU platforms," in *Proc. 28th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Västeras, Sweden, Mar. 2020, pp. 224–231.

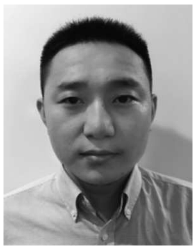
[11] R. Nozal, J. L. Bosque, and R. Beivide, "EngineCL: Usability and performance in heterogeneous computing," *Future Gener. Comput. Syst.*, vol. 107, pp. 522–537, Jun. 2020.

[12] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *Proc. 2017 IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Austin, TX, USA, Feb. 2017, pp. 27–38.

[13] T. R. W. Scogland, W. C. Feng, B. Rountree, and B. R. D. Supinski, "CoreTSAR: Core task-size adapting runtime," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 2970–2983, Nov. 2015.
[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency Comput., Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.
[15] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "Self-adaptive OmpSs tasks in heterogeneous environments," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process. (IPDPS)*, May 2013, pp. 138–149.
[16] R. K and N. N. Chiplunkar, "A survey on techniques for cooperative CPU-GPU computing," *Sustain. Computing: Informat. Syst.*, vol. 19, pp. 72–85, Sep. 2018.
[17] K. Moren and D. Göhringer, "Automatic mapping for OpenCL-programs on CPU/GPU heterogeneous platforms," in *Proc. 18th Int. Conf. Comput. Sci. (ICCS)*, Wuxi, China, Jun. 2018, pp. 301–314.
[18] Y. Che, C. Xu, and Z. Wang, "Load balancing a multi-block grids-based application on heterogeneous platform," in *Proc. IEEE 23rd Int. Conf. Comput. Sci. Eng. (CSE)*, Guangzhou, China, Dec. 2020, pp. 44–49.
[19] K. Chronaki, A. Rico, M. Casas, M. Moretó, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Task scheduling techniques for asymmetric multi-core systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 2074–2087, Jul. 2017.
[20] Z. Li, Y. Zhang, A. Ding, H. Zhou, and C. Liu, "Efficient algorithms for task mapping on heterogeneous CPU/GPU platforms for fast completion time," *J. Syst. Archit.*, vol. 114, Mar. 2021, Art. no. 101936.
[21] M. Agostini, F. O'Brien, and T. Abdelrahman, "Balancing graph processing workloads using work stealing on heterogeneous CPU-FPGA systems," in *Proc. 49th Int. Conf. Parallel Process.*, Edmonton, AB, Canada, Aug. 2020, pp. 1–12.
[22] B. Pérez, E. Stafford, J. L. Bosque, and R. Beivide, "Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems," *J. Parallel Distrib. Comput.*, vol. 157, pp. 30–42, Nov. 2021.
[23] L. Wan, W. Zheng, and X. Yuan, "Efficient inter-device task scheduling schemes for multi-device co-processing of data-parallel kernels on heterogeneous systems," *IEEE Access*, vol. 9, pp. 59968–59978, 2021.
[24] M. Gowanlock and B. Karsin, "A hybrid CPU/GPU approach for optimizing sorting throughput," *Parallel Comput.*, vol. 85, pp. 45–55, Jul. 2019.
[25] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "HiWayLib: A software framework for enabling high performance communications for heterogeneous pipeline computations," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, Providence, RI, USA, Apr. 2019, pp. 153–166.
[26] T. Li, Q. Dong, Y. Wang, X. Gong, and Y. Yang, "Dual buffer rotation four-stage pipeline for CPU–GPU cooperative computing," *Soft Comput.*, vol. 23, no. 3, pp. 859–869, Feb. 2019.
[27] F. Zhang, Z. Chen, C. Zhang, A. C. Zhou, J. Zhai, and X. Du, "An efficient parallel secure machine learning framework on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2262–2276, Sep. 2021.
[28] G. Tan, C. Shui, Y. Wang, X. Yu, and Y. Yan, "Optimizing the LINPACK algorithm for large-scale PCIe-based CPU-GPU heterogeneous systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2367–2380, Sep. 2021.
[29] L.-N. Pouchet. *PolyBench: The Polyhedral Benchmark Suite*. Accessed: Jan. 12, 2021. [Online]. Available: http://polybench.sourceforge.net
[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. *Rodinia: The Rodinia Benchmark Suite*. Accessed: Jan. 16, 2021. [Online]. Available: http://www.cs.virginia.edu/rodinia/doku.php
[31] NVIDIA Corporation. *CUDA SDK*. Accessed: Jan. 19, 2021. [Online]. Available: https://developer.nvidia.com/cuda-downloads
[32] Université de Bordeaux. *StarPU Handbook*. Accessed: May 18, 2021. [Online]. Available: https://files.inria.fr/starpu/starpu-1.3.8/html
[33] Barcelona Supercomputing Center Programming Models Group. *OmpSs Specification*. Accessed: Feb. 8, 2021. [Online]. Available: https://pm.bsc.es/ftp/ompss/doc/spec
[34] T. D. Han and T. S. Abdelrahman, "HiCUDA: high-level GPGPU programming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 78–90, Jan. 2011.
[35] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP for efficient programming and tuning on GPUs," *Int. J. Comput. Sci. Eng.*, vol. 8, no. 1, pp. 4–20, Feb. 2013.
[36] OpenACC Working Group. *OpenACC 3.1 Specification*. Accessed: Jan. 8, 2021. [Online]. Available: http://www.openacc.org/specification
[37] J. Shuja, A. Gani, K. Ko, K. So, S. Mustafa, S. A. Madani, and M. K. Khan, "SIMDOM: A framework for SIMD instruction translation and offloading in heterogeneous mobile architectures," *Trans. Emerg. Telecommun. Technol.*, vol. 29, no. 4, p. e3174, Apr. 2018.
[38] J. Shuja, S. Mustafa, R. W. Ahmad, S. A. Madani, A. Gani, and M. K. Khan, "Analysis of vector code offloading framework in heterogeneous cloud and edge architectures," *IEEE Access*, vol. 5, pp. 24542–24554, 2017.

LANJUN WAN was born in Hunan, China, in 1982. He received the B.S. and M.S. degrees in computer science and technology from the Hunan University of Technology, Zhuzhou, China, in 2005 and 2009, respectively, and the Ph.D. degree in circuits and systems from Hunan University, Changsha, China, in 2016. He is currently an Assistant Professor with the School of Computer Science, Hunan University of Technology. He has published many research articles in international conferences and journals, such as *JPDC*, *CCPE*, *ParCo*, *Sensors*, and IEEE ACCESS. His research interests include high-performance computing, parallel computing, and industrial big data analysis. He serves as a Reviewer for the *JPDC*, *CCPE*, and IEEE ACCESS.

WEIHUA ZHENG was born in Guangxi, China, in 1969. He received the B.S. degree in computer science and technology from the National University of Defense Technology, Changsha, China, in 2002, the M.S. degree in computer science and technology from Xiangtan University, Xiangtan, China, in 2010, and the Ph.D. degree in computer science and technology from Hunan University, Changsha, in 2015. He is currently an Associate Professor with the College of Electrical and Information Engineering, Hunan University of Technology, Zhuzhou, China. He has published many research articles in international conferences and journals, such as *SPL*, *TCS*, and IEEE/ACM TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS. His research interests include fast Fourier transform, image processing, and parallel computing.



XINPAN YUAN was born in Hunan, China, in 1982. He received the B.S., M.S., and Ph.D. degrees in computer science and technology from Central South University, Changsha, China, in 2005, 2008, and 2012, respectively. He is currently an Associate Professor with the School of Computer Science, Hunan University of Technology, Zhuzhou, China. He has published many research articles in international conferences and journals, such as *IJNS*, *JIPS*, and *Information*. His research interests include information retrieval, natural language processing, and parallel computing.