

Received April 1, 2021, accepted April 12, 2021, date of publication April 19, 2021, date of current version April 26, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3073955

Efficient Inter-Device Task Scheduling Schemes for Multi-Device Co-Processing of Data-Parallel Kernels on Heterogeneous Systems

LANJUN WAN^{1,2}, WEIHUA ZHENG³, AND XINPAN YUAN¹

¹School of Computer Science, Hunan University of Technology, Zhuzhou 412007, China

²College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China

³College 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, in part by the National Natural Science Foundation of China under Grant 61672224, in part by the National Key Research and Development Project under Grant 2018YFB1700204 and Grant 2018YFB1003401, and in part by the Key Research and Development Project of Hunan Province under Grant 2019GK2133.

ABSTRACT Heterogeneous systems consisting of multiple multi-core CPUs and many-core accelerators have recently come into wide use, and more and more parallel applications are developed in such a heterogeneous system. To fully utilize multiple compute devices to cooperatively and concurrently execute data-parallel kernels on heterogeneous systems, a feedback-based dynamic and elastic task scheduling scheme is proposed, which can provide a better load balance, a greater device utilization, and a lower scheduling overhead by flexibly and dynamically adjusting the workload between devices during execution. The proposed method is more suitable for data-parallel kernels whose computation and data are uniformly distributed, but is less suitable for data-parallel kernels whose computation and data are non-uniformly distributed. Thus, an asynchronous-based dynamic and elastic task scheduling scheme is proposed, which can avoid device underutilization, load imbalance across devices, and frequent kernel launches, inter-device data transfers and inter-device synchronizations by dynamically adjusting the chunk size according to the performance change during runtime. A series of experiments are conducted with 8 representative parallel applications on a hybrid CPU-GPU-MIC system, the results show that the proposed two inter-device task scheduling schemes can achieve the efficient CPU-GPU-MIC co-processing of different parallel applications by effectively partitioning work across devices.


INDEX TERMS Data-parallel kernels, heterogeneous systems, many-core accelerators, multi-core CPUs, multi-device co-processing, parallel applications, task scheduling.

I. INTRODUCTION

Heterogeneous CPU-accelerator systems have recently been widely used in high-performance computing and cloud computing due to their advantages of high performance and low power consumption. Many works have focused on utilizing both CPUs and accelerators to accelerate solving a specific application, such as matrix multiplication [1], sparse matrix-vector multiplication [2], QR factorization [3], Cholesky factorization [4], branch-and-bound algorithm [5], Smith-Waterman algorithm [6], subset-sum problem [7], particle swarm optimization [8], graph processing [9], range

query [10], computational fluid dynamics [11], and atmospheric numerical simulation [12]. These works demonstrate that the CPU-accelerator co-processing yields better performance than the CPU-only execution or accelerator-only execution. However, the full utilization of all compute devices requires researchers to carefully consider the distribution of workload between devices. Therefore, it is necessary to provide a general task scheduling mechanism which supports the efficient multi-device co-processing of most parallel applications.

Recently, many studies have focused on inter-device task scheduling mechanisms for heterogeneous systems, mainly including static scheduling [13]–[17] and dynamic scheduling [18]–[37]. Static scheduling aims to statically

The associate editor coordinating the review of this manuscript and approving it for publication was Massimo Cafaro .

determine the optimal distribution of workload between devices before execution. For example, Luk *et al.* [13] built an analytical model based on offline training to find a near-optimal task partition between CPU and GPU. Grewe and O’Boyle [14] proposed a static task partition method based on predictive model and code features to efficiently partition work between CPU and GPU for OpenCL programs. Zhong *et al.* [15] proposed a data partitioning method based on functional performance model, which can effectively balance the workload of data-parallel applications on heterogeneous CPU-GPU systems. Static scheduling can well balance the workload between devices and avoid runtime scheduling overhead. However, finding the optimal workload assignment is difficult as it relies on time-consuming offline training or code analysis, and any change in the application, problem size or system configuration may require a new training run.

Dynamic scheduling aims to effectively partition work across devices during execution, which has attracted more and more attentions recently. Many researches have concentrated on dynamic scheduling strategies designed for task-parallel applications, such as work-stealing scheduling [18], speedup-based scheduling [19], locality-aware scheduling [20], feature-aware scheduling [21], load-aware scheduling [22], energy-aware scheduling [23]. Recently some dynamic scheduling strategies designed for data-parallel applications have also been proposed. For example, Belviranli *et al.* [26] proposed a two-phase dynamic self-scheduling strategy for loop iterations on heterogeneous platforms. Wang *et al.* [27] proposed a asymptotic profiling-based dynamic co-scheduling method to assign the workload to CPU and GPU. Kaleem *et al.* [28] proposed two online profiling-based dynamic scheduling schemes to split the work between CPU and GPU. Scogland *et al.* [29], [30] developed a set of adaptive chunk-based scheduling policies for data-parallel loops to split work across heterogeneous devices. Navarro *et al.* [31] proposed a novel adaptive partitioning strategy named LogFit for parallel for-loops in irregular applications, which can dynamically find a near-optimal chunk size for the GPU and CPU to maximize utilization of the GPU and avoid load imbalance. Clarke *et al.* [32] developed new dynamic load balancing algorithms based on the partial functional performance models of heterogeneous processors, which are suitable for data-intensive parallel iterative routines and different heterogeneous platforms without any restriction on the problem size. Lastovetsky *et al.* [33] proposed a performance optimization method of scientific applications on parallel platforms, which can find the optimal partition of computations between processors through the functional performance model of the data-parallel application. To support CPU-accelerator co-execution of OpenCL applications, several novel dynamic scheduling approaches [34]–[37] have been designed for data-parallel OpenCL kernels on heterogeneous CPU-accelerator systems.

Compared with static scheduling, dynamic scheduling has stronger adaptability and does not require time-consuming

offline training, but it could easily lead to load imbalance, device underutilization, and frequent kernel launches, inter-device data transfers and inter-device synchronizations. In this paper, we propose two inter-device task scheduling schemes to support the efficient multi-device co-processing of data-parallel kernels on heterogeneous CPU-accelerator systems, they can be expected to keep load balance across devices, give a greater device utilization, and avoid frequent kernel launches, inter-device data transfers and inter-device synchronizations by making dynamic and elastic workload distribution between devices during execution.

This paper makes the following main contributions:

- A feedback-based dynamic and elastic task scheduling scheme is proposed to effectively split and balance the workload between heterogeneous devices, which is more suitable for data-parallel kernels whose computation and data are uniformly distributed.
- An asynchronous-based dynamic and elastic task scheduling scheme is designed to effectively partition work across an arbitrary set of devices, which is more suitable for data-parallel kernels whose computation and data are non-uniformly distributed.
- Experiments are conducted to verify the effectiveness of the proposed task scheduling schemes, and the results show that they can efficiently support the multi-device co-processing of data-parallel kernels on heterogeneous CPU-accelerator systems.

The rest of this paper is organized as follows. Section II gives an overview of the heterogeneous system architecture. Section III describes the inter-device task scheduling schemes. Section IV analyzes the experimental results. Section V concludes this paper.

II. OVERVIEW OF THE HETEROGENEOUS SYSTEM ARCHITECTURE

This section gives a brief overview of the hardware architecture of a commonly used heterogeneous system. As shown in Fig. 1, the heterogeneous system consists of p computing devices interconnected via the PCI-E bus, including one host device (i.e., multi-core CPUs) and $p - 1$ accelerators (such as GPU or MIC). For each accelerator that participates in multi-device co-processing, we may need to transfer the required data from host memory to accelerator memory before performing the computational task assigned to the

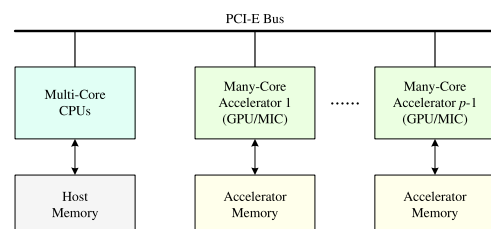


FIGURE 1. An overview of the heterogeneous system architecture.

accelerator, and we may need to transfer the results back to host memory after the accelerator has finished its work.

In this paper, the multi-device co-processing aims to fully exploit multiple compute devices of a heterogeneous system to cooperatively and concurrently execute a data-parallel kernel. However, different compute devices have different processing capability, memory capacity, and communication capability, which brings a great challenge to multi-device co-processing. The key issue of multi-device co-processing is how to effectively partition work across compute devices.

III. INTER-DEVICE TASK SCHEDULING

In this paper, a task refers to a collection of iterations within a data-parallel kernel (i.e., data-parallel for-loop). The total workload of a given task is the total number of iterations. Inter-device task scheduling aims to find the best partition of loop iterations across multiple devices. This section first discusses the previous scheduling schemes and then presents our proposed scheduling schemes.

A. PREVIOUS TASK SCHEDULING SCHEMES

This subsection discusses two simple and practical task scheduling schemes proposed by Scogland *et al.* [29], including quick scheduling and split scheduling.

1) THE QUICK SCHEDULING SCHEME

The quick scheduling scheme breaks the whole execution of a data-parallel kernel into the profiling phase and execution phase. In the profiling phase, it assigns a small portion of the total workload to each compute device according to the initial partition ratios, after each device has finished its work, it collects the execution time of each device to calculate the new partition ratios. In the execution phase, it assigns the remaining workload to each compute device according to the partition ratios calculated in the profiling phase. Moreover, if a data-parallel kernel needs to be executed many times, from the second time, the total workload is split according to the partition ratios calculated in the previous execution.

The chosen of profiling size is critical for quick scheduling. If the profiling size is too small, the partition ratios calculated in the profiling phase may not be suitable for the execution phase, this is due to the compute devices may perform differently in the two phases. In fact, the performance of one compute device may change with the workload assigned to it due to the effects of device utilization, data transfer, data race, etc. It is risky to execute a large portion of the workload once the inaccurate partition ratios are used in the execution phase. If the profiling size is too large, this could easily result in load imbalance in the profiling phase, this is because the initial partition ratios are usually not accurate enough.

By running four different benchmarks with large problem size on two Intel Xeon E5-2640v2 CPUs and an NVIDIA Tesla K40c GPU (details of the benchmarks and hardware configurations are given in Section IV-A), the performance of CPU-GPU co-processing using quick scheduling for different profiling sizes are presented in Fig. 2, where W represents the

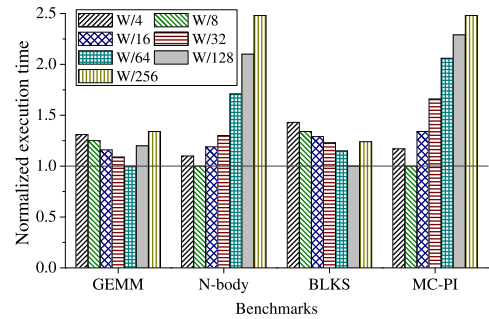


FIGURE 2. Performance of CPU-GPU co-processing using quick scheduling for different profiling sizes.

total workload of a data-parallel kernel. As shown in Fig. 2, the profiling size has much effect on the performance of quick scheduling, and the appropriate profiling size varies with different benchmarks.

2) THE SPLIT SCHEDULING SCHEME

The split scheduling scheme first splits the whole iteration space of a data-parallel for-loop into n equal-sized chunks and then uses multiple devices to cooperatively execute each chunk of iterations. Namely, it splits the total workload W into n equal parts and completes them in n steps. Specifically, in the first step, it assigns the workload of the first chunk to each compute device according to the initial partition ratios. After each device has completed its work, it collects the execution time of each device to calculate the new partition ratios. Begin from the second step, it assigns the workload of the i -th chunk to each device according to the partition ratios calculated in the previous step, where $2 \leq i \leq n$.

Although split scheduling can dynamically adjust the partition ratios to balance the workload across devices, it is sensitive to the chunk size (i.e., W/n). If the chunk size is too large, this could easily result in load imbalance because one device may have to wait for the other devices to finish execution. If the chunk size is too small, this will cause frequent kernel launches, inter-device data transfers and inter-device synchronizations. The computing power of an accelerator may also be underutilized due to the workload assigned to it is too small. Fig. 3 shows the performance of CPU-GPU co-processing using split scheduling for different chunk sizes. The results demonstrate that the performance of split scheduling is greatly affected by the chunk size, and the appropriate chunk size varies with different benchmarks.

B. THE FEEDBACK-BASED DYNAMIC AND ELASTIC TASK SCHEDULING SCHEME

Although quick scheduling and split scheduling can intelligently and adaptively partition work across devices during runtime, the performance of quick scheduling and split scheduling are sensitive to the profiling size and chunk size respectively. In this paper, we propose a novel inter-device task scheduling scheme, Feedback-based Dynamic and Elastic Task Scheduling (FDETS), which is designed to maintain load balance across devices and provide greater device

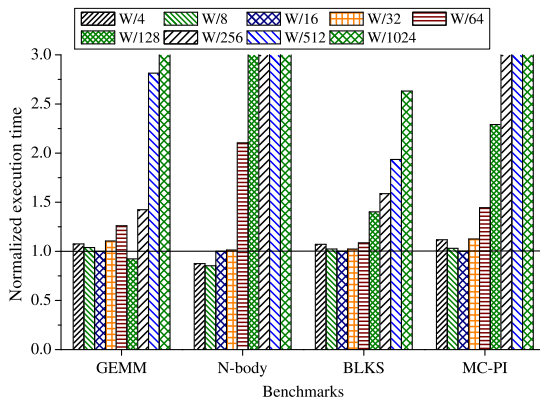


FIGURE 3. Performance of CPU-GPU co-processing using split scheduling for different chunk sizes.

utilization and lower scheduling overhead by making flexible and dynamic workload adjustments during execution.

Unlike split scheduling, FDETS dynamically splits the entire iteration space of a data-parallel for-loop into several unequal-sized chunks. It takes $1/n$ of the total number of iterations (i.e., W/n) as the initial chunk size, and the chunk size is continuously adjusted based on the observed performance during execution. To facilitate our discussion, some notations used in our proposed inter-device task scheduling schemes are listed in Table 1.

FDETS is described in Algorithm 1, which consists of the following steps.

Step 1: FDETS uses p devices to cooperatively execute the first chunk whose size is W/n . Specifically, FDETS first assigns a part of the workload of the first chunk $W_{curr.i}$ to device D_i according to the initial partition ratio R_i , where $W_{curr.i} = W_{curr} \times R_i$, $W_{curr} = W/n$, and $1 \leq i \leq p$. The initial ratios can be set statically by programmers, or they can be calculated automatically according to the theoretical peak performance of these devices. Second, FDETS executes the device-specific computational kernel on device D_i to complete the workload $W_{curr.i}$. If D_i is an accelerator, the required data need to be uploaded to D_i before execution. Third, after device D_i has completed its assigned workload, FDETS collects the current execution time $T_{curr.i}$ and calculates the current execution speed $V_{curr.i}$ of D_i , where $V_{curr.i} = W_{curr.i}/T_{curr.i}$. If D_i is an accelerator, the processed data need to be downloaded from D_i , and the data transfer time is included in the current execution time of D_i . Fourth, after all devices have finished their respective work, FDETS calculates the relative execution speed RV_i of D_i , where $RV_i = V_{curr.i}/\sum_{j=1}^p V_{curr.j}$. Here, the relative execution speeds are used as the new partition ratios, thus the partition ratios can be updated as follows: $R_i = RV_i$. Fifth, FDETS calculates the current cooperative execution speed V_{curr} , where $V_{curr} = W_{curr}/T_{curr}$ and $T_{curr} = \max(T_{curr.1}, \dots, T_{curr.p})$. Finally, FDETS updates the completed workload W_f and the remaining workload W_r , where $W_f = W_f + W_{curr}$ and $W_r = W - W_f$.

Step 2: If there is any remaining workload, FDETS uses p devices to cooperatively execute the second chunk whose size is $2 \times W/n$. Similar to Step 1, FDETS first assigns

Algorithm 1 The Feedback-Based Dynamic and Elastic Task Scheduling Scheme

Require: p , W , the initial chunk size W/n , the initial partition ratios R_1, R_2, \dots, R_p , and the threshold α

- 1: Initialize $W_f = 0$, $W_r = W$, $W_{prev} = 0$, $W_{curr} = 0$, $W_{next} = W/n$, $V_{curr} = 0$, and $j = 1$;
- 2: **while** $W_r > 0$ **do**
- 3: $W_{prev_prev} = W_{prev}$; $W_{prev} = W_{curr}$; $W_{curr} = W_{next}$;
- 4: **for** each compute device D_i , $1 \leq i \leq p$, **in parallel do**
- 5: Assign a part of the workload of the j -th chunk $W_{curr.i}$ to D_i according to the partition ratio R_i ;
- 6: Execute the device-specific computational kernel on D_i to complete the workload $W_{curr.i}$;
- 7: Collect the current execution time $T_{curr.i}$ of D_i ;
- 8: Calculate the current execution speed $V_{curr.i}$ of D_i ;
- 9: **end for**
- 10: Calculate the relative execution speed of each compute device: $RV_i = V_{curr.i}/\sum_{j=1}^p V_{curr.j}$ ($i = 1$ to p);
- 11: Update the partition ratios that will be used to split the next chunk: $R_i = RV_i$ ($i = 1$ to p);
- 12: Update the previous cooperative execution speed: $V_{prev} = V_{curr}$;
- 13: Calculate the current cooperative execution time: $T_{curr} = \max(T_{curr.1}, T_{curr.2}, \dots, T_{curr.p})$;
- 14: Calculate the current cooperative execution speed: $V_{curr} = W_{curr}/T_{curr}$;
- 15: **if** This is the first chunk, i.e., $j == 1$ **then**
- 16: $W_{next} = 2 \times W/n$;
- 17: **else**
- 18: **if** $V_{curr} > V_{prev} \times (1 + \alpha)$ **then**
- 19: **if** $W_{curr} \geq W_{prev}$ **then**
- 20: $W_{next} = W_{curr} \times 2$;
- 21: **else if** $W_{curr} < W_{prev}$ **and** $W_{curr} \geq W_{prev_prev}$ **then**
- 22: $W_{next} = W_{curr}$;
- 23: **else** $W_{next} = W_{curr}/2$; **end if**
- 24: **end if**
- 25: **if** $|V_{curr} - V_{prev}| \leq V_{prev} \times \alpha$ **then**
- 26: **if** $W_{curr} == W_{prev}$ **and** $W_{curr} \geq W_{prev_prev}$ **then**
- 27: $W_{next} = W_{curr} \times 2$;
- 28: **else** $W_{next} = W_{curr}$; **end if**
- 29: **end if**
- 30: **if** $V_{curr} < V_{prev} \times (1 - \alpha)$ **then**
- 31: **if** $W_{curr} < W_{prev}$ **then**
- 32: $W_{next} = W_{curr} \times 2$;
- 33: **else if** $W_{curr} > W_{prev}$ **and** $j == 2$ **then**
- 34: $W_{next} = W_{prev}/2$;
- 35: **else** $W_{next} = W_{curr}/2$; **end if**
- 36: **end if**
- 37: **end if**
- 38: Update the completed workload: $W_f = W_f + W_{curr}$;
- 39: Update the remaining workload: $W_r = W - W_f$;
- 40: **if** $W_{next} > W_r$ **or** $W_r - W_{next} \leq W_{curr}$ **then**
- 41: $W_{next} = W_r$;
- 42: **end if**
- 43: $j = j + 1$;
- 44: **end while**

TABLE 1. Notations used in our inter-device task scheduling schemes.

Notation	Description
p	the number of compute devices that participate in multi-device co-processing on a heterogeneous system
D_i	the i -th compute device ($i = 1$ to p)
R_i	the proportion of the workload assigned to device D_i , $\sum_{j=1}^p R_j = 1$
W	the total workload of a given task, i.e., the total number of iterations of a data-parallel for-loop
W_f	the total workload that has been completed
W_r	the remaining workload, $W_r = W - W_f$
$W_{\text{prev_prev}}$	the workload that has been completed in the one before the previous step
W_{prev}	the workload that has been completed in the previous step
W_{curr}	the workload that needs to be completed in the current step
$W_{\text{curr},i}$	the workload assigned to device D_i in the current step
W_{next}	the workload that will need to be completed in the next step
$W_{\text{next},i}$	the workload assigned to device D_i in the next step
$W_{\text{next_next}}$	the workload that will need to be completed in the one after the next step
$T_{\text{curr},i}$	the time the device D_i takes to complete its assigned workload $W_{\text{curr},i}$ in the current step
T_{curr}	the time all the devices take to cooperatively complete the workload W_{curr} in the current step, $T_{\text{curr}} = \max(T_{\text{curr},1}, T_{\text{curr},2}, \dots, T_{\text{curr},p})$
$V_{\text{curr},i}$	the speed of device D_i to complete its assigned workload in the current step, $V_{\text{curr},i} = W_{\text{curr},i}/T_{\text{curr},i}$
RV_i	the relative execution speed of device D_i ($i = 1$ to p), $RV_i = V_{\text{curr},i} / \sum_{j=1}^p V_{\text{curr},j}$
V_{prev}	the speed of all the devices to cooperatively complete the workload W_{prev} in the previous step
V_{curr}	the speed of all the devices to cooperatively complete the workload W_{curr} in the current step, $V_{\text{curr}} = W_{\text{curr}}/T_{\text{curr}}$

the workload of the second chunk to each device according to the partition ratios updated in the previous step. Second, FDETS executes the device-specific computational kernel on each device to complete its assigned workload. Third, after each device has completed its work, FDETS calculates the relative execution speed of each device and updates the partition ratios that will be used to split the next chunk. Fourth, FDETS calculates the current cooperative execution speed according to the collected information. Fifth, FDETS adjusts the size of the chunk to be executed next; i.e., it determines the workload that will need to be completed in the next step. By comparing the previous cooperative execution speed V_{prev} and the current one V_{curr} and comparing the previous chunk size W_{prev} and the current one W_{curr} , FDETS makes a decision about whether the next chunk size W_{next} should be doubled, unchanged or halved compared to the current one W_{curr} . The details of adjusting the chunk size are given in lines 18-36 of Algorithm 1. Finally, FDETS updates the completed and remaining workload.

Step 3: FDETS repeats Step 2 until the remaining workload has been completed.

As can be seen from Algorithm 1, FDETS continuously adjusts the size of the next chunk according to the dynamic changes of cooperative execution speed and workload, but W_{next} should not more than W_r . Moreover, considering that a smaller chunk may result in device underutilization at the end of the entire iteration space, FDETS calculates the difference

between W_r and W_{next} in every step, if the difference is less than or equal to W_{curr} , then $W_{\text{next}} = W_r$, otherwise W_{next} remains unchanged.

It can also be seen from Algorithm 1 that the setting of threshold α may affect the performance of FDETS. The default value of α is set to 0.1 in our experiments, and the results show that the setting is reasonable but not necessarily optimal for different benchmarks (see Section IV-E). Programmers can manually tune the value of α , and it is suggested to adjust α from 0 to 1 at an interval of 0.01, 0.02, 0.03, 0.04, or 0.05.

In addition, for some applications such as K-means clustering, a data-parallel kernel may need to be executed repeatedly, meaning that there is a need for repeated execution of the schedule. Generally, each run of the kernel consists of the several steps described above. One difference, however, is that starting with the second run of the kernel, the initial partition ratios and the sizes of the first and second chunks are determined in the previous execution of the kernel. Specifically, FDETS finds a chunk executed at the fastest speed from the previous execution of the kernel, the partition ratios corresponding to that chunk are used as the initial partition ratios, and the size of that chunk is used as the sizes of the first and second chunks.

C. THE ASYNCHRONOUS-BASED DYNAMIC AND ELASTIC TASK SCHEDULING SCHEME

Comparing to quick scheduling and split scheduling, our proposed FDETS has many advantages, but it has the following drawbacks: (i) the synchronization overhead between devices may be relatively large for data-parallel kernels that need to be executed repeatedly many times; (ii) FDETS is more suitable for data-parallel kernels whose computation and data are uniformly distributed, but is less suitable for data-parallel kernels whose computation and data are non-uniformly distributed, this is mainly because the partition ratios updated in the previous execution may not suitable for the current execution. To avoid the disadvantages of FDETS, we propose another new inter-device task scheduling scheme, Asynchronous-based Dynamic and Elastic Task Scheduling (ADETS).

Similar to FDETS, ADETS also takes $1/n$ of the total number of iterations of a data-parallel for-loop as the initial chunk size, and continuously adjusts the chunk size according to the performance change during runtime. Unlike FDETS, ADETS assigns a chunk of iterations to one compute device once it becomes idle. In other words, once the compute device has finished its work, the next unassigned chunk is assigned to it immediately. ADETS is described in Algorithm 2, which consists of the following steps.

Step 1: ADETS firstly assigns a chunk whose size is W/n to device D_i and updates the remaining workload W_r , where $W_r = W - W_f$, $W_f = W_f + W_{\text{curr},i}$, $W_{\text{curr},i} = W/n$, and $1 \leq i \leq p$. Then, ADETS executes the device-specific computational kernel on device D_i to complete the workload $W_{\text{curr},i}$. After device D_i has finished its assigned

workload, ADETS collects the execution time of D_i to calculate its execution speed $V_{curr.i}$, where $V_{curr.i} = W_{curr.i}/T_{curr.i}$.

Step 2: If there is remaining workload, similar to Step 1, ADETS firstly assigns the next unassigned chunk whose size is W/n to device D_i immediately and updates the remaining workload W_r , where $1 \leq i \leq p$. Then, ADETS executes the device-specific computational kernel on device D_i to finish the workload $W_{curr.i}$. After D_i has finished its assigned workload, ADETS collects the execution time of D_i to calculate its execution speed. In order to avoid device underutilization and reduce the overhead caused by frequent kernel launches and inter-device data transfers, ADETS calculates the variance of the previous and current execution speeds of D_i , and adjusts the size of the chunk that will be assigned to D_i in the next step according to the variance. Specifically, if $V_{curr.i} > V_{prev.i} \times (1 + \alpha)$, the chunk size will be doubled, i.e., $W_{next.i} = W_{curr.i} \times 2$, where $0 \leq \alpha < 1$. If $|V_{curr.i} - V_{prev.i}| \leq V_{prev.i} \times \alpha$, the chunk size will remain unchanged, i.e., $W_{next.i} = W_{curr.i}$. If $V_{curr.i} < V_{prev.i} \times (1 - \alpha)$, the chunk size will be halved, i.e., $W_{next.i} = W_{curr.i}/2$.

Step 3: Repeat Step 2 until all the remaining workload has been completed. In every step, the size of the chunk assigned to device D_i is determined in the previous step, once device D_i has finished its assigned workload, ADETS continues to adjust the size of the chunk that will be assigned to it in the next step according to the change of its execution speed.

To avoid the load imbalance caused by assignment of large chunks to slower compute devices at the end of the entire iteration space, in every step, ADETS checks whether the remaining workload W_r is less than or equal to $\sum_{j=1}^p W_{curr.j}$ before assigning work to each device. If $W_r \leq \sum_{j=1}^p W_{curr.j}$, ADETS assigns the next unassigned chunk whose size is $W_r \times RV_i$ to device D_i , where $RV_i = V_i / \sum_{j=1}^p V_j$. If device D_i has completed its current work, then V_i is its current execution speed $V_{curr.i}$; otherwise, V_i is its previous execution speed $V_{prev.i}$. If $W_r > \sum_{j=1}^p W_{curr.j}$, the sizes of the first and second chunks assigned to device D_i are W/n , and the size of each subsequent chunk assigned to D_i is adjusted according to the variance between $V_{prev.i}$ and $V_{curr.i}$, but the size of each chunk should not exceed $W_r - \sum_{j=1}^p W_{curr.j}$.

Moreover, if a data-parallel kernel needs to be executed repeatedly, starting with the second run of the kernel, ADETS finds a chunk executed at the fastest speed from the previous execution of the kernel, and the size of that chunk is used as both the initial chunk size and the size of the second chunk.

Finally, the performance of FDETS and ADETS are related to the initial chunk size, but our experiments verify that the initial chunk size has little impact on performance as long as it is not too small or too large (see Section IV-D). Generally, we can take a small portion of the total workload W as the initial chunk size, such as $W/16$, $W/32$, $W/64$, $W/128$, etc.

IV. EXPERIMENTAL EVALUATION

A. EXPERIMENTAL SETUP

Our experiments are carried out on a hybrid CPU-GPU-MIC system, which consists of two Intel Xeon 8-core

Algorithm 2 The Asynchronous-Based Dynamic and Elastic Task Scheduling Scheme

Require: p , W , the initial chunk size W/n , the threshold α

- 1: Initialize $W_f = 0$, $W_r = W$, $W_{next.i} = W/n$, and $V_{curr.i} = 0$ ($i = 1$ to p);
- 2: **for** each compute device D_i , $1 \leq i \leq p$, **in parallel do**
- 3: **while** $W_r > 0$ **do**
- 4: $W_{curr.i} = W_{next.i}$;
- 5: Assign a chunk whose size is $W_{curr.i}$ to device D_i ;
- 6: Update the completed workload: $W_f = W_f + W_{curr.i}$;
- 7: Update the remaining workload: $W_r = W - W_f$;
- 8: Execute the device-specific computational kernel on device D_i to complete the workload $W_{curr.i}$;
- 9: Collect the current execution time $T_{curr.i}$ of D_i ;
- 10: Update the previous execution speed of D_i :
 $V_{prev.i} = V_{curr.i}$;
- 11: Calculate the current execution speed of D_i :
 $V_{curr.i} = W_{curr.i}/T_{curr.i}$;
- 12: **if** $W_r \leq \sum_{j=1}^p W_{curr.j}$ **then**
- 13: Calculate the relative execution speed of D_i :
 $RV_i = V_i / \sum_{j=1}^p V_j$;
- 14: $W_{next.i} = W_r \times RV_i$;
- 15: **else**
- 16: **if** This is the first step **then**
- 17: $W_{next.i} = W/n$;
- 18: **else**
- 19: **if** $V_{curr.i} > V_{prev.i} \times (1 + \alpha)$ **then**
- 20: $W_{next.i} = W_{curr.i} \times 2$;
- 21: **end if**
- 22: **if** $|V_{curr.i} - V_{prev.i}| \leq V_{prev.i} \times \alpha$ **then**
- 23: $W_{next.i} = W_{curr.i}$;
- 24: **end if**
- 25: **if** $V_{curr.i} < V_{prev.i} \times (1 - \alpha)$ **then**
- 26: $W_{next.i} = W_{curr.i}/2$;
- 27: **end if**
- 28: **end if**
- 29: **if** $W_{next.i} > W_r - \sum_{j=1}^p W_{curr.j}$ **then**
- 30: $W_{next.i} = W_r - \sum_{j=1}^p W_{curr.j}$;
- 31: **end if**
- 32: **end if**
- 33: **end while**
- 34: **end for**

E5-2640v2 CPUs (16 cores at 2.0GHz), 64GB host memory, an NVIDIA Tesla K40c GPU (2880 CUDA cores at 745MHz, 12GB GDDR5 memory), and an Intel Xeon Phi 7110P Coprocessor (61 cores at 1.1GHz, 8GB memory).

Table 2 lists 8 representative benchmarks: GEMM, N-body, BLKS, CG, and MC-PI are from NVIDIA CUDA SDK [38]; K-means, LUD, and BFS are from Rodinia benchmark suite [39]. Each benchmark contains one or more data-parallel kernels, and they cover different kernel executing characteristics. For example, GEMM has one data-parallel kernel that needs to be executed one time and

TABLE 2. Benchmarks used in our experiments.

Benchmark	Abbr.	Kernel Invocation	Distribution Pattern of Data	Input Problem Size		
General Matrix Multiply	GEMM	single invocation	uniform	Small: 4k×4k	Medium: 8k×8k	Large: 12k×12k (matrix size)
K-means Clustering	K-means	multiple invocations	uniform	Small: 10M	Medium: 25M	Large: 50M (random points)
N-body Simulation	N-body	single invocation	uniform	Small: 128k	Medium: 256k	Large: 512k (bodies)
Black-Scholes Option Pricing	BLKS	multiple invocations	uniform	Small: 100M	Medium: 150M	Large: 200M (options)
Conjugate Gradient Method	CG	multiple invocations	non-uniform	Small: 4k×4k	Medium: 6k×6k	Large: 8k×8k (matrix size)
LU Decomposition	LUD	multiple invocations	non-uniform	Small: 6k×6k	Medium: 9k×9k	Large: 12k×12k (matrix size)
Monte-Carlo-PI	MC-PI	multiple invocations	uniform	Small: 200M	Medium: 400M	Large: 600M (random points)
Breadth-First Search	BFS	multiple invocations	non-uniform	Small: 4M	Medium: 8M	Large: 16M (nodes)

whose computation and data are uniformly distributed, and LUD has two data-parallel kernels that need to be executed many times and whose computation and data are non-uniformly distributed.

For each benchmark, we consider three different problem sizes and only measure the execution time of data-parallel kernels, including kernel computation time, data transfer time, kernel launch time, synchronization time, and scheduling overhead. The memory allocation and deallocation time, initialization time, and data preparation time are excluded.

To evaluate the effectiveness of our proposed inter-device task scheduling schemes, we implement these 8 benchmarks with the following nine different methods on the hybrid CPU-GPU-MIC system: 16-core CPU-only execution, GPU-only execution, MIC-only execution, CPU-GPU-MIC co-processing with static scheduling, quick scheduling, split scheduling, LogFit, FDETS, and ADETS.

B. COMPARISON OF FIVE DIFFERENT INTER-DEVICE DYNAMIC SCHEDULING SCHEMES

This subsection gives the performance comparison among five different inter-device dynamic scheduling schemes for different benchmarks on the hybrid CPU-GPU-MIC system. These scheduling schemes include quick scheduling [29], split scheduling [29], LogFit [31], and our proposed FDETS and ADETS. Given that quick scheduling is sensitive to the profiling size and split scheduling is sensitive to the chunk size, we manually choose the suitable profiling size and chunk size for each benchmark, respectively. For our proposed FDETS and ADETS, the initial chunk size is set to 1/128 of the total workload and the threshold α is set to 0.1 for each benchmark. Although we can manually find the optimal settings for each benchmark with different problem sizes, this is time-consuming and the adaptive advantage of the two dynamic scheduling schemes is lost.

Fig. 4 presents the speedups of CPU-GPU-MIC co-processing using five different scheduling schemes over the 16-core CPU-only execution for eight different benchmarks with three different problem sizes. The results show that the CPU-GPU-MIC co-processing is much faster than the 16-core CPU-only execution for most benchmarks. For example, the speedup of CPU-GPU-MIC co-processing using FDETS over the CPU-only execution is up to 3.94×, 7.94×, 2.30×, and 6.04× for GEMM, N-body, BLKS, and MC-PI, respectively. The speedup of CPU-GPU-MIC co-processing

using ADETS over the CPU-only execution is up to 3.23×, 5.39×, 3.67×, and 4.93× for K-means, CG, LUD, and BFS, respectively. The performance gain is mainly due to the full utilization of the computing power of the hybrid CPU-GPU-MIC system. Logically speaking, the more that the computing resources of a heterogeneous system are utilized, the better is the performance of multi-device co-processing. However, the data transfer between CPU and GPU/MIC can easily become the performance bottleneck of CPU-GPU-MIC co-processing for some benchmarks, such as K-means.

As shown in Fig. 4, the CPU-GPU-MIC co-processing using our proposed FDETS or ADETS performs better than that using quick scheduling or split scheduling. For example, compared with split scheduling, FDETS achieves an average speedup of 1.20×, 1.15×, 1.30×, and 1.35× for K-means, N-body, CG, and BFS, respectively. Compared with quick scheduling, ADETS achieves an average speedup of 1.15×, 1.18×, 1.32×, and 1.13× for GEMM, BLKS, LUD, and MC-PI, respectively. The results from Fig. 4 also show that our proposed FDETS and ADETS outperform LogFit in the CPU-GPU-MIC co-processing of most benchmarks. Specifically, FDETS yields the performance improvements of 5.95%, 8.12%, 7.95%, and 8.16% on average than LogFit for GEMM, N-body, BLKS, and MC-PI respectively, and ADETS achieves the performance improvements of 9.12% and 6.53% on average than LogFit for K-means and LUD respectively. The performance improvement is attributed to the fact that FDETS and ADETS can keep load balance across devices, provide a greater device utilization and reduce the overhead caused by frequent kernel launches, inter-device data transfers and inter-device synchronizations by making flexible and dynamic workload distribution between devices according to the performance change during execution.

In Fig. 4, it is noticeable that LogFit gives an average improvement in performance of 5.06% and 5.18% over ADETS for CG and BFS respectively, which shows that LogFit is more suitable to cope with some fine-grained and irregular parallel applications (such as CG and BFS) than our proposed scheduling schemes. This is mainly because LogFit provides a more sophisticated workload distribution scheme taking into account the irregularity of the workload. Actually, how to maximize the performance of multi-device co-processing for some irregular applications is still a challenge. Therefore, in future work, we will develop a more efficient scheduling scheme especially designed for the

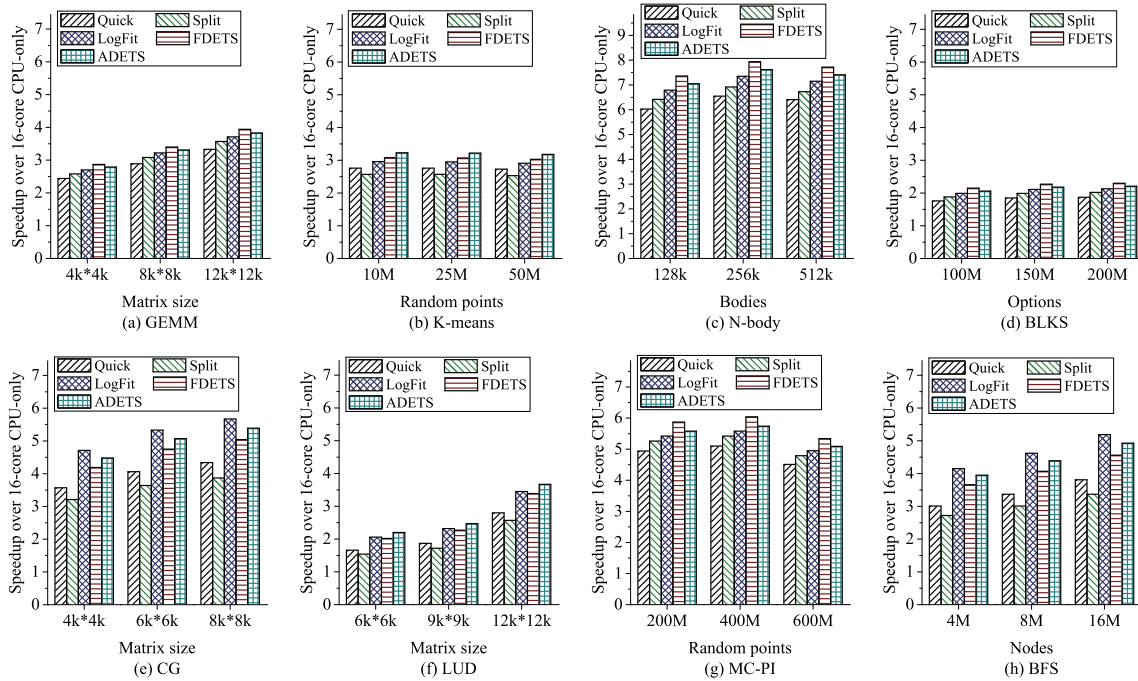


FIGURE 4. Performance comparison among five different dynamic scheduling schemes for different benchmarks on the hybrid CPU-GPU-MIC system.

multi-device co-processing of irregular data-parallel applications on heterogeneous CPU-accelerator systems.

As also can be seen from Fig. 4, for GEMM, N-body, BLKS, and MC-PI, FDETS achieves an average improvement in performance of 2.93%, 4.25%, 4.16%, and 5.13% over ADETS, respectively. However, for K-means, CG, LUD, and BFS, ADETS achieves an average improvement in performance of 4.92%, 6.92%, 8.58%, and 7.98% over FDETS, respectively. The results show that FDETS is more suitable for data-parallel kernels whose computation and data are uniformly distributed and which only need to be executed once or several times (such as GEMM, N-body, BLKS, and MC-PI), while ADETS is more suitable for data-parallel kernels that need to be executed many times and/ or whose computation and data are non-uniformly distributed (such as K-means, CG, LUD, and BFS).

C. COMPARISON WITH STATIC SCHEDULING

This subsection compares the performance of our proposed dynamic scheduling schemes with that of static scheduling. In the case of static scheduling, we manually find the near-optimal distribution of workload among CPU, GPU and MIC for each benchmark through offline training. Specifically, we first find the near-optimal partition ratios before execution and then assign the workload to CPU, GPU and MIC with the partition ratios.

Fig. 5 presents the performance comparison among static scheduling, FDETS and ADETS for different benchmarks with large problem size. As seen in Fig. 5, FDETS and ADETS perform slightly worse than static scheduling for the following benchmarks: GEMM, K-means, N-body,

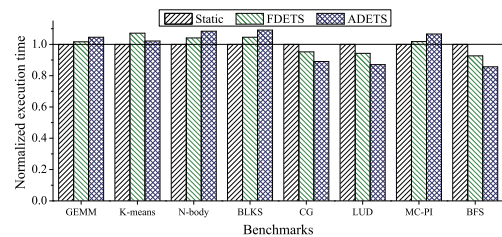


FIGURE 5. Performance comparison among static scheduling, FDETS, and ADETS for different benchmarks with large problem size.

BLKS, and MC-PI. These benchmarks have one or more data-parallel kernels whose computation and data are uniformly distributed. Compared with static scheduling, the execution time of CPU-GPU-MIC co-processing using FDETS and ADETS are increased by an average of 3.83% and 6.15%, respectively.

Fig. 5 also presents that FDETS and ADETS perform better than static scheduling for the following benchmarks: CG, LUD, and BFS. These benchmarks have one or more data-parallel kernels whose computation and data are non-uniformly distributed and that need to be executed many times. Compared with static scheduling, the execution time of CPU-GPU-MIC co-processing using FDETS and ADETS are reduced by an average of 5.97% and 12.75%, respectively. For these benchmarks, it is very difficult to find a near-optimal workload assignment that is suitable for every run of a data-parallel kernel for static scheduling.

In general, static scheduling is suitable for some data-parallel kernels that need to be executed once or several times and whose computation and data are uniformly distributed, but it requires a time-consuming offline training

to find the best partition ratios, and any change in the application, problem size or system configuration may require a new training run, as the best partition ratios are likely to change under new conditions. The poor partition ratios may cause load imbalance, this will degrade the overall performance of multi-device co-processing. Fig. 6 demonstrates this by running GEMM and K-means on two Intel Xeon E5-2640v2 CPUs and an NVIDIA Tesla K40c GPU. In Fig. 6, the x -axis shows the percentage of work allocated to the GPU is varied from 0% to 100%, and the y -axis shows the normalized execution time. The results show that the best partition differs by applications and problem sizes for static scheduling.

Compared with static scheduling, our proposed FDETS and ADETS can adapt to different applications, problem sizes and system configurations without requiring any offline training, while having lower runtime scheduling overhead.

D. IMPACT OF THE INITIAL CHUNK SIZE

Considering the initial chunk size may have an impact on performance, we evaluate the performance of CPU-GPU-MIC co-processing using our proposed FDETS and ADETS with three different initial chunk sizes: $W/64$, $W/128$, and $W/256$, where W is the total workload of a data-parallel kernel within a benchmark.

Figs. 7 and 8 present a performance comparison of three different initial chunk sizes used in our proposed FDETS and ADETS for different benchmarks with large problem size, respectively. The results show that there is a small variance in the performance achieved among different settings for GEMM, N-body, BLKS, and MC-PI. The variance becomes smaller for K-means, CG, LUD, and BFS, because these four benchmarks have one or more data-parallel kernels that need to be executed many times, whereas the initial chunk size only affects the first run of a kernel. The results verify that the performance of FDETS and ADETS are related but not sensitive to the initial chunk size. A relatively small initial chunk size is preferable for some data-parallel kernels, but if it is too small, the performance of FDETS and ADETS will be degraded because the utilization of many-core accelerator is limited for smaller workload.

E. IMPACT OF THE THRESHOLD α

In view of the setting of threshold α may also have an impact on performance, we evaluate the performance of CPU-GPU-MIC co-processing using our proposed FDETS and ADETS with ten different settings of threshold α , where α is varied from 0.05 to 0.50 at 0.05 intervals.

Fig. 9 presents a performance comparison of ten different settings of threshold α used in FDETS for three different benchmarks with large problem size. The results show that the setting of α will affect the performance of FDETS. The mean square error is 1.0909, 0.8489, and 0.7680 for GEMM, K-means, and CG, respectively. Fig. 10 presents a performance comparison of five different settings of threshold α used in ADETS for different benchmarks with large problem

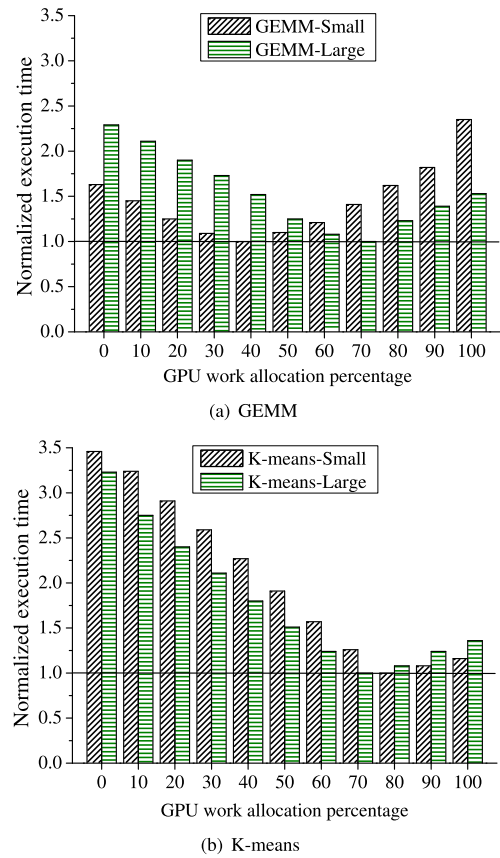


FIGURE 6. Performance of CPU-GPU co-processing using static scheduling for two different benchmarks with small and large problem sizes.

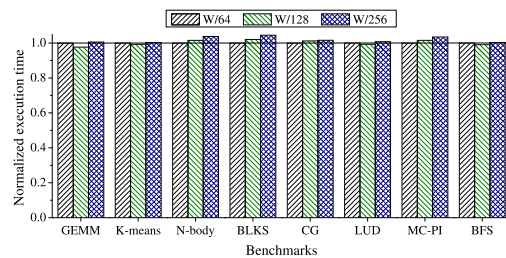


FIGURE 7. Performance comparison of three different initial chunk sizes used in our proposed FDETS for different benchmarks with large problem size.

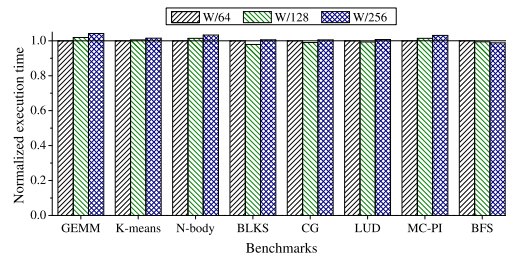


FIGURE 8. Performance comparison of three different initial chunk sizes used in our proposed ADETS for different benchmarks with large problem size.

size. It can be found from Fig. 10 that the different settings of threshold α have a little impact on the performance of

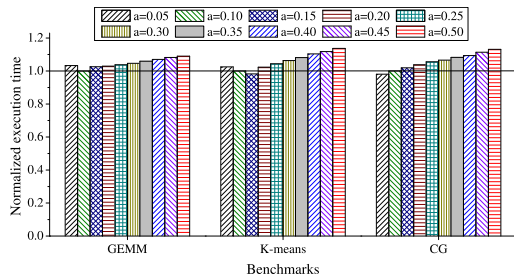


FIGURE 9. Performance comparison of ten different settings of threshold α used in our proposed FDETS for different benchmarks with large problem size.

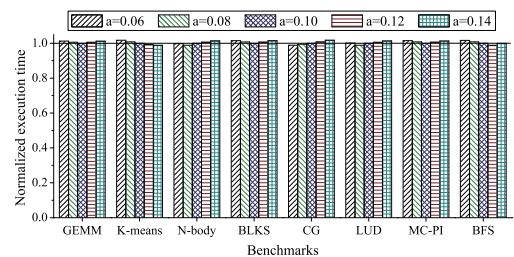


FIGURE 10. Performance comparison of five different settings of threshold α used in our proposed ADETS for different benchmarks with large problem size.

ADETS for different benchmarks. The results demonstrate that the performance of ADETS is related but not sensitive to the threshold α .

As shown in Fig. 9, the best performance is achieved when α is set to 0.10, 0.15, and 0.05 for GEMM, K-means, and CG, respectively. As shown in Fig. 10, the best performance is achieved when α is set to 0.10, 0.14, 0.08, 0.10, 0.06, 0.08, 0.10, and 0.12 for GEMM, K-means, N-body, BLKS, CG, LUD, MC-PI, and BFS, respectively. The results illustrate that the default setting of threshold α (i.e., $\alpha = 0.10$) in our proposed FDETS and ADETS is reasonable but not necessarily optimal for different benchmarks, and programmers can manually tune the threshold α .

V. CONCLUSION

In this paper, in order to support the efficient multi-device co-processing of data-parallel kernels on heterogeneous CPU-accelerator systems, we propose two different inter-device task scheduling schemes, including FDETS and ADETS, which can provide lower scheduling overhead and higher device utilization and maintain load balance across devices by making flexible and dynamic workload adjustments according to the performance change during runtime. FDETS is more suitable for data-parallel kernels whose computation and data are uniformly distributed, while ADETS is more suitable for data-parallel kernels whose computation and data are non-uniformly distributed. By conducting experiments with eight different parallel applications on a hybrid CPU-GPU-MIC system, we found that the proposed two inter-device task scheduling schemes can effectively partition work across CPU, GPU, and MIC, and the CPU-GPU-MIC

co-processing of each application significantly outperforms the CPU-only execution.

Anyone who has worked in load balancing knows that there is no definitive solution to the problem, because the optimal workload partition differs by applications, platforms, problem sizes and so on. For example, LogFit [31] outperforms our proposed scheduling schemes in two parallel applications (i.e. CG and BFS). In [32], the authors developed advanced dynamic load balancing algorithms based on partial functional performance models of heterogeneous processors, which are applicable to data-intensive iterative applications on heterogeneous parallel platforms. Our proposed scheduling schemes don't take into account the advanced functional performance models. It is worth to integrate the partial functional performance models into our scheduling schemes to further improve the performance of multi-device co-processing in the next work.

Moreover, for some data-parallel applications, inter-device communication can easily become the performance bottleneck of multi-device co-processing. In future work, we will explore the inter-device communication optimization methods to further improve the performance of multi-device co-processing.

REFERENCES

- [1] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems," in *Proc. 26th ACM Int. Conf. Supercomput. (ICS)*, Jun. 2012, pp. 365–376.
- [2] J. Nie, C. Zhang, D. Zou, F. Xia, L. Lu, X. Wang, and F. Zhao, "Adaptive sparse matrix-vector multiplication on CPU-GPU heterogeneous architecture," in *Proc. 3rd High Perform. Comput. Cluster Technol. Conf.*, Jun. 2019, pp. 6–10.
- [3] R.-B. Chen, Y. M. Tsai, and W. Wang, "Adaptive block size for dense QR factorization in hybrid CPU-GPU systems via statistical modeling," *Parallel Comput.*, vol. 40, nos. 5–6, pp. 70–85, May 2014.
- [4] J. Chen and Z. Chen, "Cholesky factorization on heterogeneous CPU and GPU systems," in *Proc. 9th Int. Conf. Frontier Comput. Sci. Technol.*, Aug. 2015, pp. 19–26.
- [5] I. Chakroun, N. Melab, M. Mezma, and D. Tuytens, "Combining multi-core and GPU computing for solving combinatorial optimization problems," *J. Parallel Distrib. Comput.*, vol. 73, no. 12, pp. 1563–1577, Dec. 2013.
- [6] 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.
- [7] L. Wan, K. Li, J. Liu, and K. Li, "Efficient CPU-GPU cooperative computing for solving the subset-sum problem," *Concurrency Comput., Pract. Exper.*, vol. 28, no. 2, pp. 492–516, Feb. 2016.
- [8] M. P. Wachowiak, M. C. Timson, and D. J. DuVal, "Adaptive particle swarm optimization with heterogeneous multicore parallelism and GPU acceleration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2784–2793, Oct. 2017.
- [9] S. Zhou and V. K. Prasanna, "Accelerating graph analytics on CPU-FPGA heterogeneous platform," in *Proc. 29th Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Oct. 2017, pp. 137–144.
- [10] 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.
- [11] Y.-X. Wang, L.-L. Zhang, W. Liu, X.-H. Cheng, Y. Zhuang, and A. T. Chronopoulos, "Performance optimizations for scalable CFD applications on hybrid CPU+MIC heterogeneous computing system with millions of cores," *Comput. Fluids*, vol. 173, pp. 226–236, Sep. 2018.
- [12] W. Xue, C. Yang, H. Fu, X. Wang, Y. Xu, J. Liao, L. Gan, Y. Lu, R. Ranjan, and L. Wang, "Ultra-scalable CPU-MIC acceleration of mesoscale atmospheric modeling on Tianhe-2," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2382–2393, Aug. 2015.

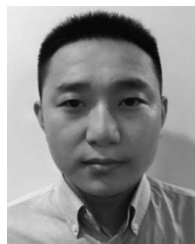
- [13] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit. (Micro)*, Dec. 2009, pp. 45–55.
- [14] D. Grewe and M. F. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *Proc. 20th Int. Conf. Compil. Const. (CC)*, Saarbrücken, Germany, Apr. 2011, pp. 286–305.
- [15] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2506–2518, Sep. 2015.
- [16] R. K and N. N. Chiplunkar, "A survey on techniques for cooperative CPU-GPU computing," *Sustain. Comput., Informat. Syst.*, vol. 19, pp. 72–85, Sep. 2018.
- [17] S. Sandokji and F. Eassa, "Task scheduling frameworks for heterogeneous computing toward exascale," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 10, pp. 234–243, 2018.
- [18] J. V. F. Lima, T. Gautier, V. Danjean, B. Raffin, and N. Maillard, "Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures," *Parallel Comput.*, vol. 44, pp. 37–52, May 2015.
- [19] Y. Wen, Z. Wang, and M. F. P. O'Boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *Proc. 21st Int. Conf. High Perform. Comput. (HiPC)*, Dec. 2014, pp. 1–10.
- [20] O. S. Simsek, A. Drebes, and A. Pop, "Leveraging data-flow task parallelism for locality-aware dynamic scheduling on heterogeneous platforms," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 540–549.
- [21] P. Du, Z. Sun, H. Zhang, and H. Ma, "Feature-aware task scheduling on CPU-FPGA heterogeneous platforms," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun.; IEEE 17th Int. Conf. Smart City; IEEE 5th Int. Conf. Data Sci. Syst. (HPC/SmartCity/DSS)*, Aug. 2019, pp. 534–541.
- [22] J. Fang, J. Zhang, S. Lu, and H. Zhao, "Exploration on task scheduling strategy for CPU-GPU heterogeneous computing system," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2020, pp. 306–311.
- [23] X. Liu, P. Liu, L. Hu, C. Zou, and Z. Cheng, "Energy-aware task scheduling with time constraint for heterogeneous cloud datacenters," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 18, Sep. 2020, Art. no. e5437.
- [24] Z. Zhu, J. Zhang, J. Zhao, J. Cao, D. Zhao, G. Jia, and Q. Meng, "A hardware and software task-scheduling framework based on CPU+FPGA heterogeneous architecture in edge computing," *IEEE Access*, vol. 7, pp. 148975–148988, 2019.
- [25] B. Wang, Y. Song, J. Cao, X. Cui, and L. Zhang, "Improving task scheduling with parallelism awareness in heterogeneous computational environments," *Future Gener. Comput. Syst.*, vol. 94, pp. 419–429, May 2019.
- [26] M. E. Belviranli, L. N. Bhuyan, and R. Gupta, "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, p. 57, Jan. 2013.
- [27] Z. Wang, L. Zheng, Q. Chen, and M. Guo, "CPU+GPU scheduling with asymptotic profiling," *Parallel Comput.*, vol. 40, no. 2, pp. 107–115, Feb. 2014.
- [28] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated GPUs," in *Proc. 23rd Int. Conf. Parallel Architectures Compilation (PACT)*, Aug. 2014, pp. 151–162.
- [29] T. R. W. Scogland, B. Rountree, W.-C. Feng, and B. R. de Supinski, "Heterogeneous task scheduling for accelerated OpenMP," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 144–155.
- [30] T. R. W. Scogland, W.-C. Feng, B. Rountree, and B. R. de Supinski, "CoreTSAR: Core task-size adapting runtime," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 2970–2983, Nov. 2015.
- [31] A. Navarro, F. Corbera, A. Rodriguez, A. Vilches, and R. Asenjo, "Heterogeneous parallel_for template for CPU-GPU chips," *Int. J. Parallel Program.*, vol. 47, no. 2, pp. 213–233, Apr. 2019.
- [32] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 195–217, Jun. 2011.
- [33] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on intel xeon phi through load imbalancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 787–797, Mar. 2017.
- [34] R. Nozal, B. Pérez, and J. L. Bosque, "Towards co-execution of massive data-parallel OpenCL kernels on CPU and Intel Xeon Phi," in *Proc. 17th Int. Conf. Comput. Math. Methods Sci. Eng. (CMMSE)*, Cádiz, Spain, Jul. 2017, pp. 1561–1572.
- [35] R. Nozal, B. Pérez, J. L. Bosque, and R. Bevide, "Load balancing in a heterogeneous world: CPU-xeon phi co-execution of data-parallel kernels," *J. Supercomput.*, vol. 75, no. 3, pp. 1123–1136, Mar. 2019.
- [36] B. Pérez, E. Stafford, J. L. Bosque, R. Bevide, S. Mateo, X. Teruel, X. Martorell, and E. Ayguadé, "Auto-tuned OpenCL kernel co-execution in OmpSs for heterogeneous systems," *J. Parallel Distrib. Comput.*, vol. 125, pp. 45–57, Mar. 2019.
- [37] M. Damschen, F. Mueller, and J. Henkel, "Co-scheduling on fused CPU-GPU architectures with shared last level caches," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2337–2347, Nov. 2018.
- [38] NVIDIA Corporation. *NVIDIA CUDA SDK Code Samples*. Accessed: Nov. 25, 2019. [Online]. Available: <https://developer.nvidia.com/cuda-downloads>
- [39] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.



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*, *Parallel Computing*, *Sensors*, and *IEEE ACCESS*. He also serves as a Reviewer for the *JPDC*, *CCPE*, and *IEEE ACCESS*. His research interests include industrial big data analysis, industrial equipment fault diagnosis, high-performance computing, and parallel computing.



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 *TCBB*. His research interests include fast Fourier transform, audio signal processing, 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 papers in international conferences and journals, such as *IJNS*, *JIPS*, and *Information*. His research interests include industrial big data analysis, industrial equipment fault diagnosis, information retrieval, data mining, and natural language processing.