Journal of Systems Architecture 114 (2021) 101936

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Efficient algorithms for task mapping on heterogeneous CPU/GPU platforms for fast completion time

Zexin Li [a], Yuqun Zhang [a,*], Ao Ding [a], Husheng Zhou [b], Cong Liu [b]

[a] Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, Guangdong, China
[b] Department of Computer Science, The University of Texas at Dallas, TX, USA

ARTICLE INFO

Keywords:
GPU
Heterogeneous scheduling
Data-size-based prediction
Neural network runtime acceleration

ABSTRACT

In GPU-based embedded systems, the problem of computation and data mapping for multiple applications while minimizing the completion time is quite challenging due to large size of the policy space. To achieve fast competition time, a fine-grain mapping framework that explores a set of critical factors is needed for heterogeneous embedded systems. In this paper, we present a theoretical framework that yields a sub-optimal solution via three practical mapping algorithms with low time complexity. We evaluate such algorithms upon StarPU with a large set of popular benchmarks. Experimental results demonstrate that algorithms proposed by the original EMSOFT paper can achieve up to 30% faster completion time compared to state-of-the-art mapping techniques, and can perform consistently well across different workloads. We further extend such algorithms to minimize the completion time and enhance the runtime performance of complex heterogeneous applications under resource-limited infrastructure. We also extend the evaluation by deploying StarPU under multiple setups with an additional benchmark testing suite for simulating real-world runtime neural networks. Experimental results demonstrate that our extended algorithm can achieve much faster completion time (averagely 30% to 37% under multiple resource-constraint scenarios) compared to the state-of-the-art mapping techniques.

1. Introduction

Graphics processing units (GPUs) are now commonly used as co-processors in many embedded systems to accelerate general-purpose applications. They are particularly capable of executing data-parallel applications, due to their highly multi-threaded architecture and high-bandwidth memory. Various embedded system domains can benefit high performance and better energy efficiency from utilizing GPUs. For example, GPUs can efficiently perform matrix operations such as factorization on large data sets and multidimensional FFTs and convolutions. Such operations are often seen in many embedded applications including signal processing, imaging and video processing. By leveraging new programming models, such as CUDA [1] and OpenCL [2], programmers can effectively develop highly data-parallel tasks to execute such applications on GPUs.

By providing heterogeneous processing elements with different performance characteristics in the same system, heterogeneous CPU/GPU architectures are expected to provide more flexibility for better performance compared to homogeneous systems. Fast completion time is an imperative performance metric that needs to be optimized in most embedded systems. For example, in a driver-assisted and autonomous vehicle, the video streaming and sensor data processing tasks need to be completed in a rapid manner. In order to minimize the completion time for running a set of workloads, the step that maps computations to processing elements is critical. In this paper, we consider the mapping problem in a heterogeneous system containing multiple CPUs and GPUs. Our goal is to minimize the completion time.

This mapping problem is quite challenging due to a large size of the policy space. First of all, applications may demonstrate (sometimes significantly) different performance characteristics when executed on GPUs than CPUs. The mapping algorithm thus needs to consider such heterogeneity when making prioritization and mapping decisions. Moreover, most real world workloads are implemented using rather complex task graphs, where a task graph contains a number of data- or logical-dependent tasks. The precedence constraints among tasks require the mapping algorithm to consider: *(i)* the task graph structure and *(ii)* different data transfer costs among tasks if executed on different processors. Furthermore, for data-intensive tasks, data partitioning techniques need to be incorporated into the mapping algorithm because partitioning a task into threads that can be run on multiple devices in parallel improves the overall utilization.

[*] Corresponding author.
E-mail addresses: 11610515@mail.sustech.edu.cn (Z. Li), zhangyq@sustech.edu.cn (Y. Zhang), 11612521@mail.sustech.edu.cn (A. Ding), husheng.zhou@utdallas.edu (H. Zhou), cong@utdallas.edu (C. Liu).

https://doi.org/10.1016/j.sysarc.2020.101936
Received 22 January 2020; Received in revised form 9 November 2020; Accepted 11 November 2020
Available online 19 December 2020

Without considering the above-mentioned factors, mapping algorithms are unlikely to perform consistently well across different workloads. Prior work on heterogeneous CPU/GPU systems has focused on new programming models and API extensions for supporting multiple heterogeneous devices [3–5], automating the mapping processor [6–8], enabling CPU and GPU sharing [9]. Different mapping heuristics have been designed and applied in these work. However, since the fine-grain mapping problem is not the major focus of these work, the existing mapping heuristics make simplified mapping decisions based upon a limited set of metrics (e.g., data locality or execution time).

In this paper, motivated by a number of measurements-based case studies, we design three mapping algorithms, each of which explores a specific set of factors that may affect the completion time performance. We evaluated such algorithms by implementing them on a real heterogeneous system, i.e., containing a four-core CPU and two discrete GPUs with different performance characteristics. Extensive experiments were conducted using a set of popular benchmarks and workloads, such as Cholesky factorization, Monte Carlo. Experimental results demonstrate that our proposed algorithms can achieve much faster completion time (up to 30% improvement) compared to the state-of-the-art mapping techniques. By testing workloads with varying characteristics, experiments show that the completion time performance under our mapping algorithms is also consistent.

We further extend the original scheduling algorithms by including the data transfer time, i.e., reducing the time consumption for data transfer prediction. To evaluate our extended algorithm, we deployed the original StarPU in a new heterogeneous system containing two ten-core CPUs and four discrete GPUs with close setups. To further verify the performance of all proposed scheduling algorithms, we implemented an extended benchmark testing suite for simulating runtime neural network applications in real-world scenarios. Experimental results demonstrate that our extended algorithm can achieve much faster completion time (averagely 29% in more computing resource case and averagely 37% in less computing resource case) compared to the state-of-the-art mapping techniques.

The contributions of this paper are listed as follows.

- **Idea and approach.** We attempt to minimize the completion time of generalized applications in heterogeneous systems by optimizing scheduling strategies via proposing a set of algorithms including *heterogeneity ratio-based mapping algorithm*, *structure rank based heuristics algorithm* and *data partition algorithm* [10].
- **Evaluation.** We conduct a set of experiments on a real-world runtime system, e.g., StarPU to evaluate our proposed algorithms and compare their performance with naive and state-of-the-art scheduling algorithms.
- **Extended Approach.** We propose an extended algorithm, namely *heterogeneity ratio-based and data-partition optimizing scheduling*, to minimize the completion time and enhance the runtime performance of deep-neural-network-based applications under resource-limited infrastructure.
- **Extended Evaluation.** We conduct a set of experiments upon StarPU with server-level setups to evaluate the runtime performance of *heterogeneity ratio-based and data-partition optimizing scheduling* and compare it to the naive, state-of-the-art, and the originally proposed algorithms.

The rest of this paper is organized as follows. Section 2 presents the background, system model and our theoretical framework. Section 3 describes the measurement-based and extensive case studies as our motivation. Section 4 presents the practical mapping algorithms and the extended mapping algorithm for neural network acceleration. Section 5 describes our implementation. Section 6 discusses our experimental results, including the experimental results for the extended algorithm in both the original benchmarks and the extended benchmark. Section 7 describes related work. Section 8 concludes this paper.

Table 1
Notation summary.

N	Number of total tasks
n	Number of applications
m	Number of processors
τ_i	ith application
τ_i^z	zth kernel/task of application τ_i
M_i	ith processor (either a CPU or a GPU)
$C_{i,k}^j$	Execution time of τ_i^j on processor M_k
$\mathcal{S}(\tau_i^j)$	Set of successor kernels/tasks of τ_i^j
$\mathcal{P}(\tau_i^j)$	Set of predecessor kernels/tasks of τ_i^j
e_i^{jk}	Edge from τ_i^j to τ_i^k
$T_{q\rightarrow w}(e_i^{jk})$	Time taken to send data from τ_i^j to τ_i^k

2. Background

In this section we give out a list of notations and definitions to help us better formalize the proposed problem, and then we briefly describe the general structure of heterogeneous schedulers.

2.1. System model

Let us consider the problem of mapping n independent applications $\Gamma = \{\tau_1, \tau_2, \tau_3, \dots, \tau_n\}$ onto m processors $M = \{M_1, M_2, \dots, M_m\}$. Each processor is either a CPU or a GPU.

Each application τ_i is composed by serial instructions and tasks, where tasks represent computation operations. tasks of each application are chained together according to the computation logic and may have dependencies since data flows from one task to another. To be specific, τ_i is modeled as a task graph that contains z_i connected tasks $\{\tau_i^1, \tau_i^2, \dots, \tau_i^{z_i}\}$. Let N denote the total number of tasks of applications in Γ. Each task τ_i^j has an execution time of $C_{i,k}^j$ if executed on processor p_k. The execution time ranges from milliseconds to hours, depending on the specific application. Similar to prior work [3], we use the sampling functionality of StarPU [11] to obtain the estimated execution time of a task.

Between any two connected tasks is an edge, which implies that precedence constraints exist between these two tasks. If task τ_i^j has an outgoing edge e_i^{jk} to task τ_i^k, then τ_i^k cannot start execution until it receives the data produced by τ_i^j. Let $\mathcal{P}(\tau_i^j)$ denote the set of predecessor tasks of τ_i^j, i.e., tasks that have outgoing edges to τ_i^j. Similarly, let $\mathcal{S}(\tau_i^j)$ denote the set of successor tasks of τ_i^j, i.e., tasks that have incoming edges to τ_i^j. Let $T_{q\rightarrow w}(e_i^{jk})$ denote the time for τ_i^j executed on processor p_q to send its produced data to its successor τ_i^k (connected by edge e_i^{jk}) executed on processor p_w. A summary of important notation is given in Table 1. We use the term *task* to represent a task combining with its needed data. For readability, in the rest of this paper, we will use *task* and *task* interchangeably.

Definition 1. We define the depth of a task to be the number of tasks on the longest path between this task and a task of the corresponding task graph that has no predecessors. tasks with no predecessors have a depth of 1. Let $D(\tau_i^z)$ denote depth of task τ_i^z in the task graph of τ_i.

Preemptive vs. non-preemptive execution On GPUs, executions are often non-preemptive [12]. That is, once a task starts execution on a GPU, it cannot be preempted by other tasks until its completion. On CPUs, executions can be preemptive. However, preemptions may incur significant amount of overheads at runtime such as context switch overhead and migration overhead [13]. To ensure the efficiency, as well as simplify the formalism and algorithms, we thus assume in this paper non-preemptive executions on CPU as well.

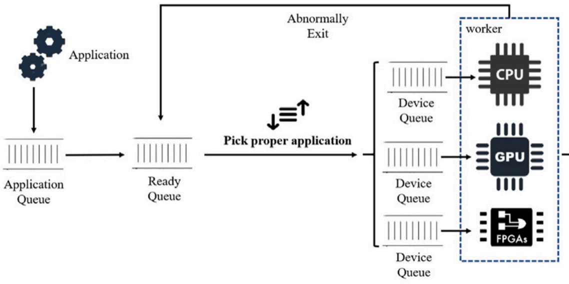

Fig. 1. General structure of heterogeneous schedulers.

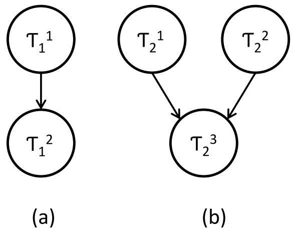

Fig. 2. Task dependency graph.

2.2. General structure of heterogeneous schedulers

As shown in Fig. 1, the general structure of a heterogeneous scheduler is implemented upon three types of queues: the application queue, $ReadyQueue$ and device queue. Initially, the to-be-executed applications are assigned to the application queue. Secondly, applications get into the $ReadyQueue$ to be ready for execution. Then, proper applications are selected in the $ReadyQueue$ and push into the device queues corresponding to different devices. Finally, the applications exit the queues under the terminated executions while they are pushed back to the $ReadyQueue$ for the subsequent scheduling process under abnormally terminated executions.

3. Case studies: What to consider for making mapping decisions

In this section, we present several measurements-based case studies that motivate the design of our mapping algorithms. We measured the completion time of executing a vector add application τ_1(τ_i is the ith application) and a matrix multiplication application τ_2 on a heterogeneous system configured with one Intel Core i7 CPU and NVIDIA GeForce GTX660 GPU. τ_1 can be expressed as $(v_1 + v_2) * \pi$, where v_1 and v_2 are vectors and π is a constant. τ_2 can be expressed as $(a * b) + (c * d)$, where a, b, c, d are four input matrices. These applications are commonly seen in scientific computing. The corresponding task graphs are illustrated in Fig. 2. Specifically, τ_1 contains two tasks τ_1^1(τ_i^z is the zth task/task of application τ_i) and τ_1^2, where τ_1^1 is a vector add task and τ_1^2 is a vector scale task. τ_2 contains three tasks τ_2^1, τ_2^2, and τ_2^3, where τ_2^1 and τ_2^2 are two matrix multiplication tasks, and τ_2^3 is a matrix add task. For the generated input data, v_1 and v_2 have a size of 50000 elements each. a, b, c, and d are four matrices with a size of $1024 * 1$, $1 * 1024$, $1024 * 1024$, and $1024 * 1024$, respectively. Through profiling, the execution time of each task is listed in Table 2. We have conducted various experiments based upon this system setup and recorded the corresponding mapping sequences and completion time under different strategies. Among the obtained results, we have identified several factors that may significantly affect the mapping performance.

Observation #1: task-level mapping or application-level mapping? In this case study, our observation is that for applications that contain multiple dependent tasks, treating tasks as the mapping entity yields better performance than mapping each entire application to a processing unit. Fig. 3(a) shows the schedule of performing application-level mapping. The dash lines in this figure represent the final completion time. The (tiny) space among task execution blocks represents the delay due to necessary data transfer. Fig. 3(b) shows the schedule of performing task-level mapping, in which we can see that the completion time is shortened. The main performance acceleration comes from the parallel executions of multiple tasks on two processing units. Intuitively, for

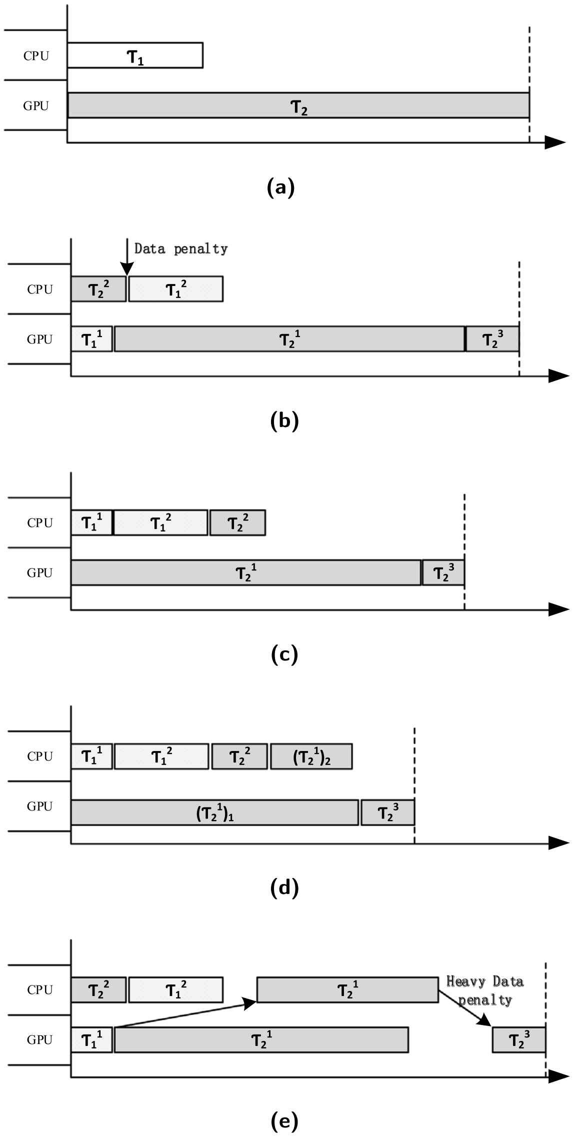

Fig. 3. (a) Application level mapping (b) task level mapping (c) Different map order (d) Data Partition (e) Bad data partition.

Table 2
Execution time of tasks.

	CPU	GPU
τ_1^1	5.68×10^2 µs	4.22×10^2 µs
τ_1^2	1.52×10^3 µs	2.42×10^2 µs
τ_2^1	4.41×10^4 µs	5.6×10^3 µs
τ_2^2	8.74×10^2 µs	8.44×10^2 µs
τ_2^3	4.40×10^2 µs	4.20×10^2 µs

systems that support multiple applications, task-level mapping is a better choice because it can better utilize the hardware resources.

Observation #2: heterogeneity matters. Fig. 3(c) shows the schedule of a task mapping policy with a different task ordering scheme than the mapping policy shown in Fig. 3(b). The applied mapping policy considered in this case prioritize tasks by considering the heterogeneity. Intuitively, a task that has a faster execution time on a specific type of processor (either CPU or GPU) should preferably be assigned to that type of processor. As shown in Table 2, tasks τ_1^2, and τ_2^1 have much shorter execution time on GPU compared to CPU. Thus, by prioritizing such tasks over other tasks (such as τ_1^1 and τ_2^2), they have higher possibilities to be assigned to their favorite processors, as observed in Fig. 3(c). This case study highlights the fact that for CPU/GPU systems, the heterogeneity reflected by hardware and application characteristics must be considered in the mapping algorithm.

Observation #3: data partitioning—is it always beneficial? As seen in Table 2, τ_2^1 is the most computation-intensive task. Fig. 3(c) shows that τ_2^3 cannot start execution because τ_2^1 completes late, which causes resource under-utilization and longer completion time. By partitioning the input matrix of τ_2^1 into two slices, we are able to reduce its execution time by running the task with partial data on both CPU and GPU in parallel. Let $(\tau_2^1)_1$ and $(\tau_2^1)_2$ denote the resulting two tasks each with half data. The resulting schedule with reduced completion time is shown in Fig. 3(d). However, data partitioning is not free. It incurs additional data transfer overhead because data need to be sent to both $(\tau_2^1)_1$ and $(\tau_2^1)_2$, and the corresponding results need to be merged and then sent to τ_2^3. Since the data size is not very large in this case, the performance gain due to data partitioning overwhelms the penalty due to additional data transfer. Nevertheless, when we increase the input matrix size for τ_2^1 to 16384 * 16384, the negative impact due to additional data transfer under partitioning becomes obvious, as illustrated in Fig. 3(e). Our observation herein is that data partitioning may be beneficial only when the input data size is reasonably small.

In summary, it is clear from these case studies that the completion time performance heavily depends on the mapping algorithm, which needs to consider a number of influential factors including the task structure, heterogeneity, task prioritization, and data partitioning.

4. Practical mapping algorithms

In this section, we present three practical online algorithms for mapping tasks in a heterogeneous platform consisting of multiple CPUs and GPUs. Our algorithmic design is motivated by the observations as discussed in Section 3. Specifically, the proposed mapping algorithms consider heterogeneity, task graph structure, and data partitioning. The first algorithm (we call it the baseline algorithm) mainly factors heterogeneity into making mapping decisions (besides considering traditional factors such as data locality and earliest completion time). The second algorithm considers task structure when prioritizing tasks. The third algorithm extends the baseline algorithm by taking advantages of data partitioning. As seen in Section 6, these three algorithms yield different performance under different experimental scenarios, depending on specific application characteristics.

4.1. Baseline algorithm: Heterogeneity ratio-based mapping

As discussed in Section 3, without considering heterogeneous workload characteristics on CPUs and GPUs, the mapping algorithm is unlikely to efficiently utilize the heterogeneous resources. Our proposed baseline algorithm takes heterogeneity into consideration when making mapping decisions. Before describing the algorithm, we first give several definitions.

Definition 2. The **favorite ratio** $F_{i,k}^j$ of a task τ_i^j executed on processor M_k is defined to be

$$F_{i,k}^j = \frac{max_{h=1}^m (C_{i,h}^j)}{C_{i,k}^j} \tag{1}$$

For any task τ_i^j, a larger $F_{i,k}^j$ value implies τ_i^j is more suitable to be executed on M_k. That is, τ_i^j may have a shorter execution time if executed on M_k compared to other processors.

Definition 3. The **heterogeneity ratio** of a task τ_i^j is defined to be

$$H_i^j = max_{k=1}^m (F_{i,k}^j) \tag{2}$$

For any task τ_i^j, a large heterogeneity ratio implies that it may be more beneficial to execute τ_i^j on one of its favorite processors M_k where $F_{i,k}^j$ is large.

Example. Considering the example system described in Section 3, the *favorite ratio* of τ_1^1 (when τ_1^1 is executed on processor 1 (CPU)) is $F_{1,1}^1 = max(C_{1,1}^1, C_{1,2}^1)/C_{1,1}^1 = 5.68/5.68 = 1$, and the favorite ratio of τ_1^1 (when τ_1^1 is executed on processor 2 (GPU)) is $F_{1,2}^1 = max(C_{1,1}^1, C_{1,2}^1)/C_{1,2}^1 = 5.68/4.22 = 1.35$. The heterogeneity ratio of τ_1^1 can be calculated by $H_1^1 = max(F_{1,1}^1, F_{1,2}^1) = F_{1,2}^1 = 1.35$.

Definition 4. Let $MDAC(\tau_i^j, M_q)$ denote the **Max Data Transfer Time** of τ_i^j if τ_i^j is assigned on M_q, which is defined as the maximum time for transferring data from any of τ_i^j's predecessor tasks to τ_i^j. Specifically, $MDAC(\tau_i^j, M_q)$ is given by

$$MDAC(\tau_i^j, M_q) = \max_{\tau_i^k \in \mathcal{P}(\tau_i^j)} T_{g \to q}(e_i^{kj}) \tag{3}$$

where tau_i^k is executed on M_g.

Definition 5. Let $EFT(\tau_i^j, M_q)$ denote the **Earliest Finish Time** of τ_i^j if τ_i^j is assigned on M_q. It is defined as:

$$EFT(\tau_i^j, M_q) = T_{Avail}(M_q) + MDAC(\tau_i^j, M_q) + C_{i,q}^j \tag{4}$$

where $T_{Avail}(M_q)$ is the earliest time at which processor M_q is available,

In Definition 5, we define $T_{Avail}(M_q)$ as the earliest time at which processor M_q is available. Each device is pre-calibrated before running so that we can know exactly how long per fine-grained tasks run on them. And we can know the number and type of the remaining fine-grained tasks in the device queue in a given device. Based on such information, we can infer $T_{Avail}(M_q)$.

Our proposed baseline algorithm prioritizes tasks based on their heterogeneity ratio. The intuition is to give tasks with larger heterogeneous ratios higher possibilities to be assigned on their favorite processing units. Computing each task's heterogeneity ratio at runtime may incur a considerable amount of overheads. To avoid such overheads, in our implementation, we maintain a lookup table for each task, which records its historical sampling information. For instance, for the matrix multiplication task, each entry in the lookup table contains data size, average execution time, processing unit to which it is assigned, heterogeneity ratio, hash value, etc. Thus, at runtime, we only need to check the lookup table to figure out the needed information (e.g., heterogeneity ratio). After prioritizing tasks, the algorithm selects the best processing unit for executing each task in turn based on the earliest finish time. The pseudo-code of the algorithm is given in Algorithm 1.

Pseudo-code description. The *PushTask()* function on Line 1 is in charge of pushing incoming tasks into the *ReadyQueue* of the scheduler. It first obtains the heterogeneity ratios from the lookup table for each incoming task (Line 2), then inserts the tasks into the *ReadyQueue* by largest-heterogeneity-ratio-first (Lines 3–8). On Line 9, function *GetAllDeviceLen()* gets the total number of assigned tasks in all device

Algorithm 1 Heterogeneity ratio-based mapping

```
1:  function PushTask($task$)
2:      Sort tasks in the $ReadyQueue$ by largest-heterogeneity-ratio-first
3:      for $t_i$ in $ReadyQueue$ decreased by $H_i$ do
4:          if $H(task) < H(t_i)$ then
5:              continue
6:          end if
7:          $InsertBefore(task, t_i, ReadyQueue)$
8:      end for
9:      $num \leftarrow GetAllDeviceLen()$
10:     if $num < thr$ then
11:         PushTaskOnDeviceQueue
12:     end if
13: end function
14:
15: function PushTaskOnDeviceQueue
16:     $\tau_i^j \leftarrow PopFront(ReadyQueue)$
17:     for $M_q$ in processor set $M$ do
18:         $EFT(\tau_i^j, M_q) = T_{Avail}(M_q) + MDAC(\tau_i^j, M_q) + C_{\tau_i^j, M_q}$
19:     end for
20:     Assign $\tau_i^j$ to $M_q$ that minimize $EFT(\tau_i^j, M_q)$
21: end function
```

queues. If the number is less than a predefined threshold *thr*(Line 10), then the scheduler executes the *PushTaskOnDeviceQueue()* function. In other words, if the total number of tasks that have been assigned to devices is large enough, then the scheduler will stop dispatching tasks in the $ReadyQueue$ to devices. The intuition is to let the $ReadyQueue$ hold most of the unassigned tasks and sort them in order while guaranteeing that processing units have enough tasks residing in their device queues to be executed. Unlike the greedy dispatching approach that assigns ready tasks immediately to devices, our non-greedy approach ensures that tasks entering the $ReadyQueue$ late still have a fairly good chance to be assigned to their favorite processing units. The function *PushTaskOnDeviceQueue* (Lines 15–21) seeks to assign tasks to devices. It first grabs the task with highest heterogeneous ratio (Line 16), then estimates the finish time of this task if assigned to each processor (Lines 17–19), and finally assigns the task to the processor that yields the earliest finish time (Line 20).

Time complexity. This algorithm need to compute the heterogeneity ratio and conduct sort insertion that is $O(l^2)$, and the assignment phase needs $O(l^2 \cdot m)$ time complexity. The total time complexity is $O(l^2 \cdot m)$ where l is the maximum number of tasks in the $ReadyQueue$ and m is the number of processors.

Rationale behind threshold setup For time complexity, if we keep m (the number of processors) as constant, the time complexity of Algorithm 1 will only depend on l (the maximum number of tasks) in the $ReadyQueue$. If the $ReadyQueue$ cannot dequeue or slowly dequeue, but rapidly enqueue, the time consumption of computation of heterogeneity ratio and sort insertion will rapidly increase. To address this issue, the threshold is used in Algorithm 1 to limit the maximum capacity of the $ReadyQueue$. Therefore, the maximum extra time consumption of the algorithm itself can be ensured under a preset constraint, i.e., $O(thr^2 \cdot m)$.

4.2. Task graph structure considerations

Our second algorithm improves upon the baseline algorithm by considering the task graph structure of each application. As discussed in Section 3, our observation is that for many applications, the time taken to transfer data among tasks executed on different devices (which heavily depends on the task graph structure) is far from being negligible when compared to task execution time. For certain data-intensive applications, the data transfer time is actually the dominant factor in response time performance. Let $T(e_i^{jk})$ represent the general data transfer cost between two dependent tasks t_i^j and t_i^k. Since $T(e_i^{jk})$ can be decided only after knowing the specific devices to which these two tasks are assigned, we compute the average cost as the estimated data transfer time between t_i^j and t_i^k, which is given by

$$T(e_i^{jk}) = \frac{\sum_{q,w\in M}\left(T_{q\to w}(e_i^{jk})\right)}{m^2}. \tag{5}$$

Note that if τ_i^j and τ_i^k are assigned to the same device, then $T(e_i^{jk}) = 0$.

The algorithm seeks to assign higher priorities to tasks with larger $rank(\tau_i^j)$ values. $rank(\tau_i^j)$ is defined as:

$$\begin{aligned} rank(\tau_i^j) = & \sum_{M_q\in M} C_{i,q}^j/m + \max_{\tau_i^k\in S(\tau_i^j)} \Big(T(e_i^{jk}) \\ & + \sum_{M_q\in M} C_{i,q}^k/m \Big), \end{aligned} \tag{6}$$

where $\sum_{M_q\in M} C_{i,q}^j/m$ denotes the average execution time of task τ_i^j, and the max() term represents the longest time taken to send τ_i^j's data to any of its successor tasks plus this successor's execution time. The intuition behind using $rank(\tau_i^j)$ values is to give pairs of connected tasks that are computation-intensive and/or data-intensive higher possibilities to be assigned to their favorite devices. The pseudo-code of this algorithm is given in Algorithm 2. As seen, the algorithm is identical to our baseline algorithm except that the scheduling priorities the tasks using the $rank(\tau_i^j)$ values instead of the heterogeneity ratios.

Algorithm 2 Structure rank based heuristics

```
function PushTask($task$)
    $rank(task) \leftarrow$ Compute $rank$ of $task$
    for $t_i$ in $ReadyQueue$ decreased by $rank(t_i)$ do
        if $rank(task) < rank(t_i)$ then
            continue
        end if
        $InsertBefore(task, t_i, ReadyQueue)$
    end for
    $num \leftarrow GetAllDeviceLen()$
    if $num < thr$ then
        PushTaskOnDeviceQueue
    end if
end function
```

4.3. Data partitioning

According to the observation given in Section 3, the intuition behind data partitioning is that if a task is data-intensive, then dividing its data into multiple slices would give it a higher chance to utilize more processors. This idea has been proposed and applied in [14], but only under a single task scenario. For example, an automated partitioning technique has been proposed in [3] to partition the data of a single task such that this task can be executed on a CPU and a GPU in parallel. Unlike prior work, our third algorithm considers data partitioning as a sub-component and integrates it into our considered multi-task scenario.

Despite of its advantages, data partitioning may also introduce additional data transfer costs, as discussed in Section 3. Thus, a mapping algorithm needs to decide whether to apply data partitioning to applications. Our third algorithm extends the baseline heterogeneity ratio-based algorithm by taking data partitioning into account. We apply a historical data profiling technique to decide whether a task needs to be partitioned. In the implementation, we record the historical sampling data and use a nonlinear regression-based cost model ($a * D^b + c$) [15] (where a, b, and c are constant coefficients, and D

is the data size) to find out the relationship between data size and execution time. Given the data size of a task, if the estimated execution time (without applying data partitioning) is larger than a pre-defined threshold, then we partition it into multiple blocks.

4.4. Data transfer time considerations

The priority functions of the aforementioned algorithms fail to consider or can hardly accurately capture the data transfer time. In particular, “h-ratio” considers data transfer time by using $MDAC$, but sometimes it is inaccurate since data transfer time is related to the size of data transferred where using maximum value to estimate this value tends to cause system non-optimization, i.e., the over-estimation. On the other hand, “d-rank” and “dmdar” fail to take data transfer time into considerations. However, in real-world heterogeneous computing scenarios, the data transfer time usually can cause non-negligible impacts and ought to be considered.

To address such issue, based on the previous work, we develop a nonlinear scheduling algorithm which adopts a newly-designed priority function on different processor types. In particular, since both state-of-the-art algorithms and our original algorithms, i.e., “h-ratio” and “d-rank”, lead to long prediction time under massive processors and further result in inferior runtime performance, our nonlinear scheduling algorithm replaces *MDAC* with *PDAC* (predicted data transfer time) and revokes the threshold setups.

According to Definition 4, *MDAC* calculates the time consumption for data transfer between two devices, with the time complexity of $O(m^2)$ where the number of devices is denoted by *m*. It can be observed that although the *MDAC* computation can perform well under small device number, the time consumption to calculate $MDAC$ rises rapidly when the device number largely increases (e.g., on a high-performance computing cluster, there are thousands of devices). Therefore, it is difficult to ensure the real-time property during the *MDAC* computation process.

We can observe that in various embedded platforms, e.g., StarPU [11], data transfer time is mainly dominated by the data size in transfer. Furthermore, we infer that replacing $MDAC$ with a metric derived by a nonlinear model based on data size in transfer can significantly reduce the time consumption for predicting data transfer time, because it is unlikely to compute $MDAC$ in a real-time system at a small time cost. However, the data size in transfer can be easily accessed. Therefore, we can expect to reduce scheduling time cost for massive-device computation, e.g., runtime neural networks by adopting data size in transfer in our scheduling algorithm.

Definition 6. Let $PDAC(\tau_i^j, M_q)$ denote the **Predicted Data Transfer Time** of τ_i^j if τ_i^j is assigned on M_q, which is defined as the maximum time for transferring data from any of τ_i^j’s predecessor tasks to τ_i^j. Specifically, $PDAC(\tau_i^j, M_q)$ is given by

$$PDAC(\tau_i^j, M_q) = (a * D^b + c) \tag{7}$$

where a, b and c are hyper-parameters obtained from devices, D denotes the data size in transfer.

To illustrate, considering that the time consumption for data transmission cannot be neglected in the overall computation, we decide to propose a nonlinear model ($a * D^b + c$), similar to [15] based on data size (mentioned in data partition Section 4.3) using conventional optimization methods, e.g., heuristic method, for predicting data transfer time. After training for fitting the parameters, the time complexity of the entire prediction process can be reduced to a constant level, i.e., $O(1)$, which significantly enhance the forecasting efficiency under tolerantly biased predictions.

Based on such considerations, we replace *EFT* defined in Definition 5 by *PFT* (Predicted Finish Time) via the predicted data transfer time such that the algorithm with the neural network acceleration considerations can use *PFT* to predict data transfer time more accurately.

Definition 7. Let $PFT(\tau_i^j, M_q)$ denote the **Predicted Finish Time** of τ_i^j if τ_i^j is assigned on M_q. It is defined as:

$$PFT(\tau_i^j, M_q) = T_{Avail}(M_q) + PDAC(\tau_i^j, M_q) + C_{i,q}^j \tag{8}$$

where $T_{Avail}(M_q)$ is the earliest time at which processor M_q is available,

Algorithm 3 Heterogeneity ratio-based and data-partition optimizing scheduling

```
function PushTask($task$)
    Sort tasks in the $ReadyQueue$ by largest-heterogeneity-ratio-first
    for $t_i$ in $ReadyQueue$ decreased by $H_i$ do
        if $H(task) < H(t_i)$ then
            $continue$
        end if
        $InsertBefore(task, t_i, ReadyQueue)$
    end for
    PushTaskOnDeviceQueue
end function

function PushTaskOnDeviceQueue
    $\tau_i^j \leftarrow PopFront(ReadyQueue)$
    for $M_q$ $in\ processor\ set\ M$ do
        $PFT(\tau_i^j, M_q) = T_{Avail}(M_q) + PDAC(\tau_i^j, M_q) + C_{i,q}^j$
    end for
    Assign $\tau_i^j$ to $M_q$ that minimizes $PFT(\tau_i^j, M_q)$
end function
```

Since the task-based heterogeneity features of the ReadyQueue ensures that in most cases, heterogeneous tasks can be processed with priority. Intuitively, a more heterogeneous task has a higher probability of being prioritized to be assigned to the corresponding device, such that the tasks with short execution time can be ensured for being scheduled in a high priority. However, such scheduling strategy may result in the halting states of the less heterogeneous tasks for a long time, similar to thread starvation [16]. Moreover, since the calculation of the task heterogeneity cannot fully reflect the exact execution time, it cannot always guarantee that the less heterogeneous tasks are assigned with long execution time and thus fail to maintain runtime performance optimality.

To optimize runtime performance, the original heterogeneous scheduling algorithm in Section 4.1 uses the threshold to manage the ReadyQueue (Lines 9–12 of Alg. 1). When the tasks in the DeviceQueue are excessively accumulated at any moment, the threshold-enabled ReadyQueue temporarily halts the handover operation of the multi-level queue, as it were blocking pre-scheduled tasks in the ReadyQueue, similar to busy waiting, for optimizing the runtime performance of the overall scheduling algorithm under the capacity of the DeviceQueue.

In Section 6.2.3, to implement the threshold setups in Alg. 1, getAllDevicelen() needs to be called (Line 9) where its execution time is positively correlated with the length of DeviceQueue. Therefore, it is difficult to preset a uniform threshold for optimizing runtime performance under diverse scenarios. To be specific, if one task is not placed in the DeviceQueue when the DeviceQueue length exceeds the preset threshold, no further tasks can be added into the DeviceQueue. However, the task heterogeneity still needs to be computed and thus causes computing resource inefficiency. Meanwhile, in practice, application-level strategies can be used to limit the number of simultaneous tasks pushed to the ReadyQueue, e.g., by adopting an interrupt-based scheduling mechanism (in fact, this is often a solution to performance bottlenecks for preventing memory or computing device overflow errors).

Based on the above analysis, we decide to remove the threshold setups in our extended scheduling algorithm, i.e., Lines 9–12 of Alg. 1, for optimizing time and resource consumption when scheduling tasks.

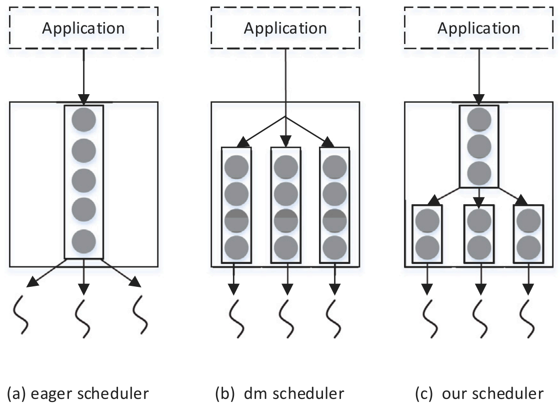

Fig. 4. Our scheduler implementation.

The details of our extended scheduling algorithm, namely *heterogeneity ratio-based and data-partition optimizing scheduling* is demonstrated in Alg. 3. The *heterogeneity ratio-based and data-partition optimizing scheduling* consists of two steps. In particular, similar to Alg. 1, the incoming tasks are first pushed into the $ReadyQueue$ of the scheduler (Lines 1–9) is in charge of pushing. Next, the *PushTaskOnDeviceQueue()* function (Lines 11–17) assigns tasks to the corresponding device, where it deques a task (denoted by τ_i^j) from $ReadyQueue$ (Line 12), calculates PFT for each device (Lines 13–15), and assigns the task to the device that minimizes PFT. In the following, we denote our heterogeneity ratio-based and data-partition optimizing scheduling as "hratio-part".

Time complexity. Similar to Alg. 1, the "hratio-part" computes the heterogeneity ratio and conducts sort insertion within $O(l^2)$, and the time complexity for the task assignment is $O(l^2 \cdot m)$. By replacing *MDAC* with *PDAC*, we reduce time complexity of finish time estimation from $O(m^2)$ to $O(1)$. As a result, the total time complexity of the overall algorithm stays $O(l^2 \cdot m)$ as the "h-ratio" and "d-rank" where l is the number of tasks and m is the number of processors.

5. Implementation

We implement our scheduler algorithms on top of the StarPU runtime platform [11] as customized schedulers. The role of the StarPU scheduler is to dispatch tasks onto different processing units (named "workers" internally). In general, the process of a scheduler can be described as follows: given n applications, each application consists of a number of tasks waiting to be executed. The scheduler selects the tasks from the runnable tasks (i.e., the tasks that obtain all the data they need) for each processing unit once the processing unit is free. Moreover, the scheduler is expected to ensure the completion time of the application to be as short as possible.

5.1. General scheduler implementation

All StarPU scheduling strategies implement task dispatching using queue-based method. Tasks that have received needed data from their predecessors are pushed in a $ReadyQueue$. This $ReadyQueue$ is updated at runtime while tasks arrive dynamically. Based upon this dispatching model, our schedulers make mapping decision at runtime for tasks in $ReadyQueue$.

StarPU has several pre-defined schedulers, including the eager scheduler, the dm scheduler, and the dmda scheduler. The eager scheduler uses a single FIFO task queue, as illustrated in Fig. 4(a), from which workers draw tasks to execute. The mapping decision is made only when a worker becomes idle. More complex schedulers such as the dm scheduler maintain one queue for each processing unit, as shown in Fig. 4(b). A task is immediately dispatched to a specific worker once it is pushed into the $ReadyQueue$. Different from these implementation strategies, our scheduler uses a central priority queue to hold and sort tasks, and dispatch tasks to worker's private queues, as illustrated in Fig. 4(c). Under our implementation, the proposed schedulers do not immediately dispatch an incoming task to one of the workers' queues. Instead, we set a threshold value (as discussed in Section 4.1) to trigger the dispatching action. The central priority queue would dispatch tasks to workers only when the total number of tasks residing in workers' queues is less than the pre-defined threshold value. A large threshold value may allow the scheduler to have a better ordering of the $ReadyQueue$. However, when considering multiple application scenarios, the total number of tasks could be large. Since pushing tasks into the $ReadyQueue$ may incur overheads, a large threshold value may also reduce the efficiency as such overheads negatively impact the timing performance. Although depending on the specific hardware, the idea behind setting a threshold value is to perform task pushing and task execution in parallel at runtime.

5.2. Implementation of *hratio-part*

As shown in Fig. 4(c), *hratio-part* is implemented by multi-level queues which makes fine-grained scheduling possible. Compared to Fig. 4(a) and (b), *hratio-part* has stronger control over the on-scheduling tasks and can perform complex classification of tasks in the first-level queue. In the implementation of *hratio-part*, $ReadyQueue$ dominates the runtime, e.g., calculating task heterogeneity and insertion sorting. Meanwhile, $DeviceQueue$ only needs to dispatch the corresponding tasks to its associated device. Next, we will discuss the implementation details in StarPU platform.

In StarPU, we modify the core code of StarPU to customize and register our scheduler to StarPU's environment. First, we bind the scheduler pointer to the structure of StarPU scheduler "predefined_policies" so that StarPU can recognize the customized scheduler. Then, we use a StarPU structure named "_starpu_sched_hratiopart_ready_policy" to specify the concrete implementation hook of the predefined scheduler interface. Finally, we implement the corresponding task handler function (implementing the interface mentioned above) for scheduling (e.g., pop task from the $ReadyQueue$, push task to device queues, etc.).

6. Evaluation

In this section, we present the implementation methodology and experimental results used to evaluate the effectiveness of our proposed algorithms.

6.1. Experimental setup

We implemented the "h-ratio" and "d-rank" in a real heterogeneous desktop computer consisting of a CPU and two discrete GPUs. The hardware specification is given in Table 4. The benchmarks used in the experiments are listed in Table 3. All benchmarks are rewritten in order to be used on the StarPU runtime platform. Among the benchmarks, Monte Carlo and Cholesky factorization are considered to be computation-intensive because they have relatively heavier computing workload for processor units and have a relatively high computation-to-communication ratio (i.e., the task execution time is far greater than the time to transfer its needed data from another device). On the other hand, VectorAdd and VectorIncrement are considered to be date-intensive because their computing workload is low, but may generate heavy date traffic. To reflect different workload scenarios, we vary the problem scale of each benchmark as three problem sizes. To further evaluate our *heterogeneity ratio-based and data-partition optimizing scheduling*, we change the original PC-level settings to server-level settings and re-implement "h-ratio" and "d-rank" with extended benchmarks to implement all the algorithms (i.e., Algs. 1 to 3) for evaluating runtime neural network applications.

Table 3
Benchmarks used in the original experiments.

Benchmark	Description	Small problem size	Medium problem size	Large problem size
xgemm	Combined matrix multiplication and addition	1k*1k matrix x 3	4k*4k matrix	8k*8k matrix
cg	Conjugate Gradient	1k*1k matrix and 1k vector	4k*4k matrix and 4k vector	8k*8k matrix and 8k vector
cholesky	Cholesky matrix factorization	1k*1k matrix	2k*2k matrix	4k*4k matrix
increment	Vector instrumentation	10k vector	100k vector	1M vector
va	Vector add	10k vector	100k vector	1Mk vector
pi	Monte Carlo method to compute pi	1k hits per task, 1k tasks	4k hits per task, 1k tasks	8k hits per task, 1k tasks
fblock	3-D assignment	128*128*128* cube	256*256*256 cube	512*512*512 cube

The specific values of the problem sizes generated in the experiments are shown in Table 3. Moreover, we test three workload composition scenarios commonly seen in practice, i.e., computation-intensive, data-intensive, and randomly mixed workloads. To generate these composition scenarios, we first generate one instance of each of the first seven benchmarks shown in Table 3 as the base case. We then generate the computation-intensive workload composition using the base case combined with three instances of each of the two computation-intensive benchmarks (mentioned above). Similarly, the data-intensive workload composition is generated using the base case combined with three instances of each of the two data-intensive benchmarks. The mixed workload composition is generated by creating two instances of each of the seven considered benchmarks. Note that the current StarPU runtime system implementation mainly considers the single application scenario. To support simultaneous execution of multiple applications, in our experiments, we compose all the benchmarks into one single executable file by rewriting and compiling the source codes of the benchmarks using StarPU's SDK.

We compare our proposed mapping algorithms against the reference scheduler of StarPU—the dmdar (deque model data aware ready) scheduler [17], which considers the task execution time and the data transfer time when making mapping decisions. According to StarPU handbook, similar to the deque model scheduler (dm scheduler), dmdar is designed by involving data transfer time (dmda scheduler), while it also uses priority-queue to distribute tasks. It is similar to the classical heterogeneous-earliest-finish-time-first scheduling (HEFT): dmdar schedules each task to a processing unit that provides the minimum finish time, and sorts tasks residing in each worker queue by largest number of available data buffers first. For each experimental setup, first, we tested two system configurations: one with one CPU and two GPUs, and the other one with one CPU and one GPU (GTX 660), as specified in Table 4. Regarding the evaluation metric, we measured the final completion time for running each entire experiment set. In the following, we denote our baseline mapping algorithm (Section 4.1), structure-based mapping algorithm (Section 4.2), data partitioning-based mapping algorithm (Section 4.3), the dmdar scheduler , as "h-ratio", "d-rank", "ad-part" and"dmdar", respectively.

Nowadays, the neural network models, e.g., GoogleNet [18] and Fast R-CNN [19,20], can achieve superior accuracy while enabling large number of layers. Correspondingly, they tend to consume excessive computing resources than traditional network models and are more likely to cause performance bottlenecks. As a result, it is difficult to apply them on resource-limited embedded systems, e.g., neural-network-based autonomous driving systems, while maintaining optimal runtime performance.

To evaluate the performance of the "h-ratio" and "d-rank" and the extended algorithm on runtime neural networks, we design and implement an extended benchmark for simulating runtime deep neural networks on StarPU [11]. In particular, we simulate the different layers of deep neural networks via their respective typical computation load and the corresponding data dependency, e.g., matrix multiplication for simulating the convolution layers and bubble sorting for simulating the interpreted layers.

For evaluating our "hratio-part" in high-performance computing setups, we redeploy the experimental environment according to the hardware specification given in Table 5 which consists of two CPUs and four discrete GPUs with much more powerful settings. Table 6 lists our extended benchmarks for simulating runtime deep neural networks (nn-LeNet, nn-GoogleNet, nn-ImageNet, nn-FRCNN) with the corresponding functions reconstructed on the StarPU runtime platform.

Table 4
Experimental hardware specification for the original evaluation.

	CPU	GPU1	GPU2
Architecture	Intel Core i7-4700	NVIDIA GeForce GTX 660	NVIDIA GeForce GT 620
Frequency	3.9 GHz	1033 MHz	700 MHz
Memory	16 GB DDR3	2048 MB GDDR5	2048 MB DDR3
OS	64-bit Linux Ubuntu lucid		

Table 5
Experimental hardware specification for the extended evaluation.

	CPU*2	GPU*4
Architecture	Intel(R) Xeon(R) CPU E5–2640	NVIDIA GeForce GTX 1080 Ti
Frequency	2.40 GHz	1480 MHz
Memory	126 GB DDR4	11264 MB GDDR5
OS	64-bit Linux Ubuntu Xenial	

6.2. Results

We conduct three sets of experiments to evaluate our original and extended algorithms. In particular, Fig. 5 shows the experimental results of the "h-ratio" and "d-rank" under the original benchmarks (Table 3) and experimental setups (Table 4). In all six graphs, the x-axis denotes the three tested scenarios where problem size scale is varied to be small, medium, and large (according to Table 3). The y-axis denotes the speedup each algorithm can achieve upon the naive CPU-only mapping algorithm. Graphs in the first (second) row depict the results under the system configuration with one CPU and two GPUs (one CPU and one GPU). In the first (respectively, second and third) column of the graphs, mixed (respectively, computation-intensive and data-intensive) workloads are assumed.

Fig. 6 shows the experimental results of the original and extended algorithms under the original benchmarks (Table 3) and the extended experimental setups (Table 5). In all the six graphs, the x-axis denotes the three tested scenarios where problem size scale varies to be small, medium, and large (according to Table 3). The y-axis denotes the achieving speedup of each algorithm upon the eager scheduler. Graphs in the first (second) row depict the results under the system configuration with two CPUs and four GPUs (two CPUs and two GPUs). In the first (respectively, second and third) column of the graphs, mixed (respectively, computation-intensive and data-intensive) workloads are assumed.

Figs. 7 and 8 show the experimental results under the extended benchmarks (Table 6) and experimental setups (Table 5).

6.2.1. Experiments under the original benchmarks and experimental setups

The obtained experimental results comparing our original mapping algorithms against dmdar are shown in Fig. 5 (the organization of which is explain in the figure's caption). Each bar

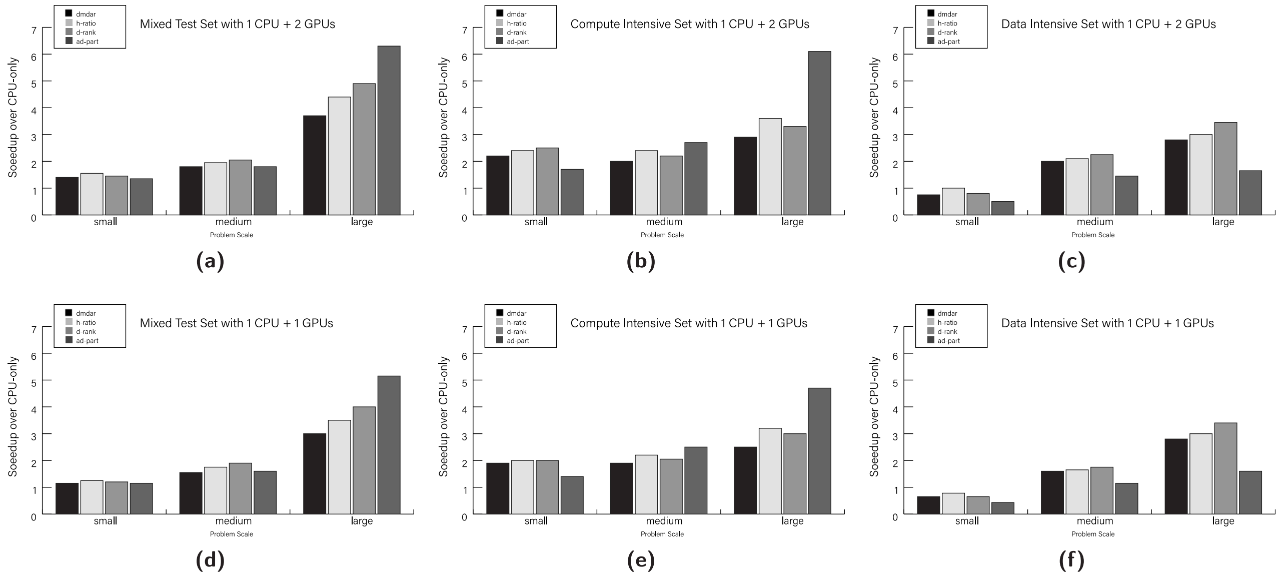

Fig. 5. Original experimental results on the original benchmarks in Table 3 with the configuration in Table 4.

Table 6
Benchmarks used in the extended experiments with configuration in the Table 4.

Benchmark	Description	Small problem size	Medium problem size	Large problem size
nn-LeNet	LeNet neural network simulation	256 tasks	1k tasks	2k tasks
nn-GoogleNet	GoogleNet neural network simulation	100 tasks	200 tasks	400 tasks
nn-ImageNet	ImageNet neural network simulation	1k tasks	4k tasks	8k tasks
nn-FRCNN	FRCNN neural network simulation	1k tasks	4k tasks	8k tasks

plots the speedup achieved by the corresponding algorithm upon a naive CPU-only mapping algorithm which prioritizes workloads by shortest-execution-time-first and maps all workloads only to CPU.

As seen, in most tested scenarios, our original mapping algorithms improve upon dmdar. The performance gain varies depending on the workload composition and problem scale. As shown in all six graphs of Fig. 5, when the problem size is small or medium, one or more of our proposed algorithms yield slightly better performance than dmdar. The improvement is not significant in these cases because the variances in heterogeneity ratio and structure spawn are small. Thus, the benefit of specifically considering these factors becomes less significant. When the problem size becomes large, the performance improvement under our proposed algorithms becomes more substantial. For example, as seen in Fig. 5(b), for computation-intensive workloads with large problem size, h-ratio, d-rank, and ad-part improve upon dmdar by more than 15%, 10%, and 110%, respectively. In particular, ad-part achieves the best performance in these cases because computation-intensive tasks are divided into parallel threads with partial data. This effectively reduces the time to complete such tasks when multiple processing units become available. Moreover, for computation-intensive tasks, applying data partitioning does not incur much data transfer penalty. Another interesting observation is that when workloads become data-intensive, ad-part yields the worst performance, as shown in Figs. 5(c) and (f). By analyzing the mapping traces of these experiments, we observe that partitioning data-intensive applications may incur significant data transfer time, which negatively impact the completion time performance. Unlike prior work considering single application scenario where data partitioning should be applied in most cases, our results suggest that data partitioning should only be selectively applied, in particularly when workloads become more data-intensive. Figs. 5(d)–(f) show the results under the system configuration with the CPU and only one GPU (removing the less powerful GT 620 GPU). Compared to the case where all three processing units are used (shown in Figs. 5(a)–(c)), the observation is that the speedup decreases. This is intuitive because less resources are available in this case.

6.2.2. *Experiments for evaluating “hratio-part”*

For evaluating our “hratio-part” as well as comparing its performance with the “h-ratio” and “d-rank”, we set up our experiments into two groups: one is implemented to evaluate their performance upon up-to-dated embedded computing device setups and the other upon high-performance computing device setups.

Overall, all our proposed algorithms, including the original and extensive ones, can achieve around 10% better completion time than the dmdar algorithm. While the original scheduling algorithms significantly outperform the eager scheduling algorithm, they fail to do so over the dmdar algorithm, possibly because the computation platforms, such as CUDA [1] and OpenCL [2], have been increasingly improved in recent years so that a simple scheduling strategy can also have a relatively acceptable effect. Another reason is that the eager scheduling algorithm for the original text comparison is only run on the CPU, and even for the Naive algorithm, GPU acceleration is allowed for a more fair comparison.

In the extended schemer “hratio-part”, because of the nonlinear relationship based on the data size to estimate the data transfer time, the time consumption cost of the scheduling is significantly reduced, which leads to significant enhancement of the real-time performance of the scheduling. For instance, in Fig. 6(a) and (d) (mixed test scenarios), all five tested schedulers (including “hratio-part”) can significantly outperform the speed of the eager scheduler as in the medium/large test scenarios, while in small size of test scenarios, the schedulers based on the dmdar/h-ratio/d-rank algorithms performs much worse than the eager scheduler. The small test scenarios require smaller time consumption of computing tasks, but close time to scheduling. This shows their scheduling time consumption cannot be ignored in small test scenarios and “hratio-part” can efficiently reduce scheduling time.

We further observe that on the benchmarks for simulating LeNet [21], GoogleNet [18,22], and ImageNet [23] (Fig. 8(a)–(c) and Fig. 7(a)–(c)), the existing algorithms (dmdar, h-ratio, d-rank) and the extended algorithm (“hratio-part”) can achieve close performance as

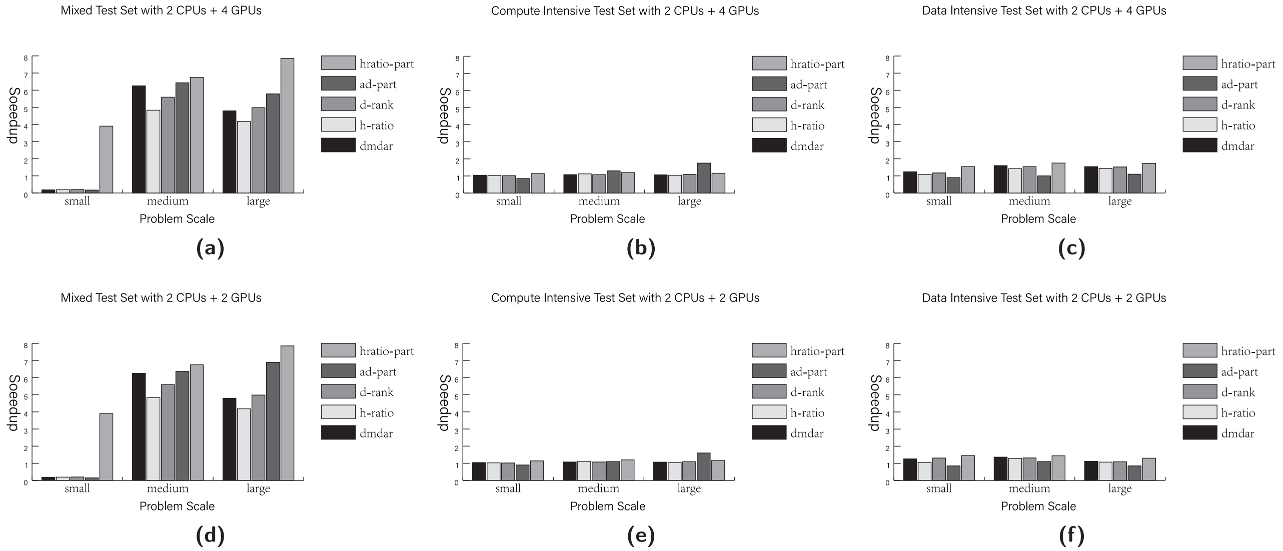

Fig. 6. Experimental results on the original benchmarks in Table 3 with the configuration in Table 5.

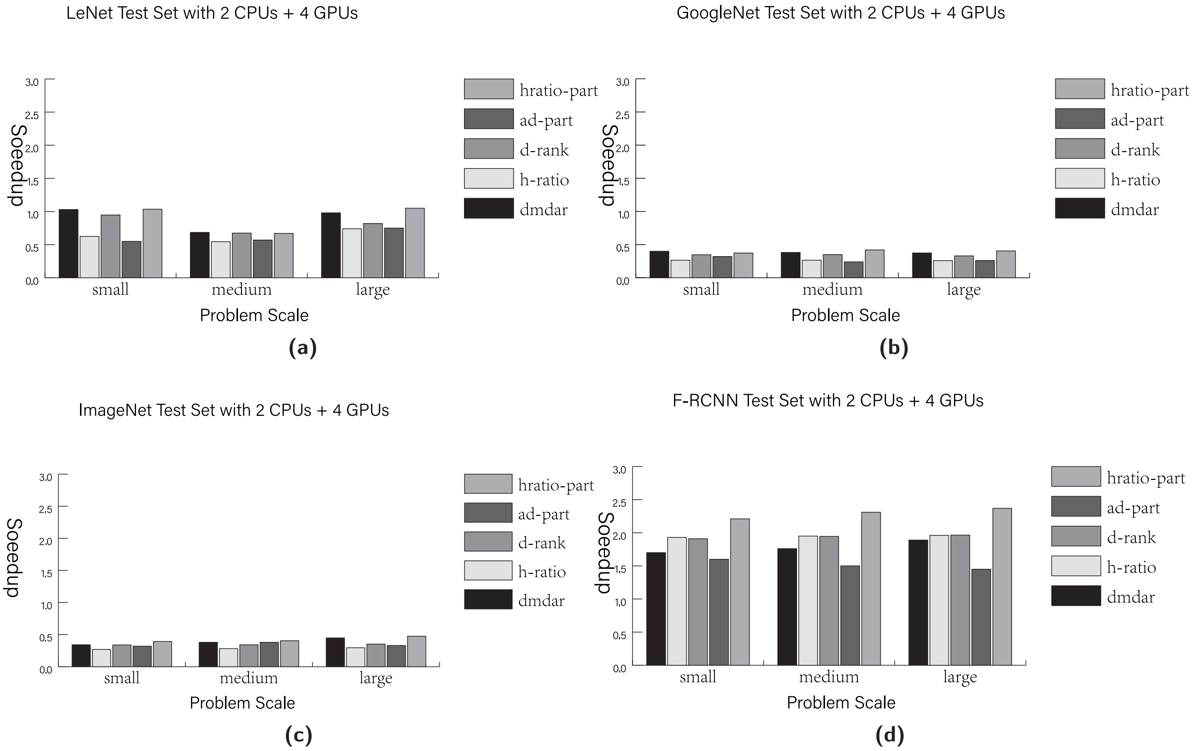

Fig. 7. Experimental results of NN simulation on 2CPUs+4GPUs (extended benchmarks in Table 6) with the configuration in Table 5.

the baseline FIFO-based algorithm, which can be explained as follows. Since the proposed algorithms tend to schedule each single task of an application, such scheduling process inevitably results in overall non-trivial time consumption all together. However, in typical deep neural networks, e.g., LeNet, GoogleNet and ImageNet, most of the size of their associated computation tasks are small [18,21–23] compared to the original benchmarks adopted in this paper (the input size of convolutional layers is usually less than 100 × 100, while the input size of original benchmarks is more than 1kx1k, as listed in Table 3). As a result, compared with the execution time for tasks, scheduling tasks consumes more time. We define the total running time, i.e., scheduling time plus execution time, as the evaluation metric of runtime performance.

On the benchmark simulating F-RCNN [19,20] (nn-FRCNN), dmdar and other heterogeneous-based scheduling algorithms can greatly enhance the performance of nn-FRCNN for simulating runtime F-RCNN network. For example, in the large test scenario of Figs. 8(d) and 7(d), dmdar, h-ratio, d-rank, and “hratio-part” outperform the eager scheduler by 53%, 132%, 131%, and 159%, respectively. The h-ratio and d-rank algorithms achieve more advantages over dmdar under 2 GPUs (averagely 49%), while such advantages are shrunk under 4 GPUs (averagely 29%). It can be speculated that there are better chances to optimize the scheduling strategy when the resources are limited (with fewer GPUs).

Moreover, since LeNet and GoogleNet enable mostly homogeneous convolution operations and pooling operations, the efficiency of such

Table 7
Crash test comparison against threshold policy.

	Exp. set 1 pi large size × 100	Exp. set 2 cg large size ×100	Exp. set 3 incrementer large size × 100	Exp. set 3 fblock large size × 100
eager (naive scheduler)	0/100	0/100	1/100	0/100
dmdar (has threshold)	2/100	4/100	3/100	0/100
h-ratio (no threshold)	1/100	9/100	2/100	0/100
rank-based (no threshold)	2/100	3/100	1/100	0/100

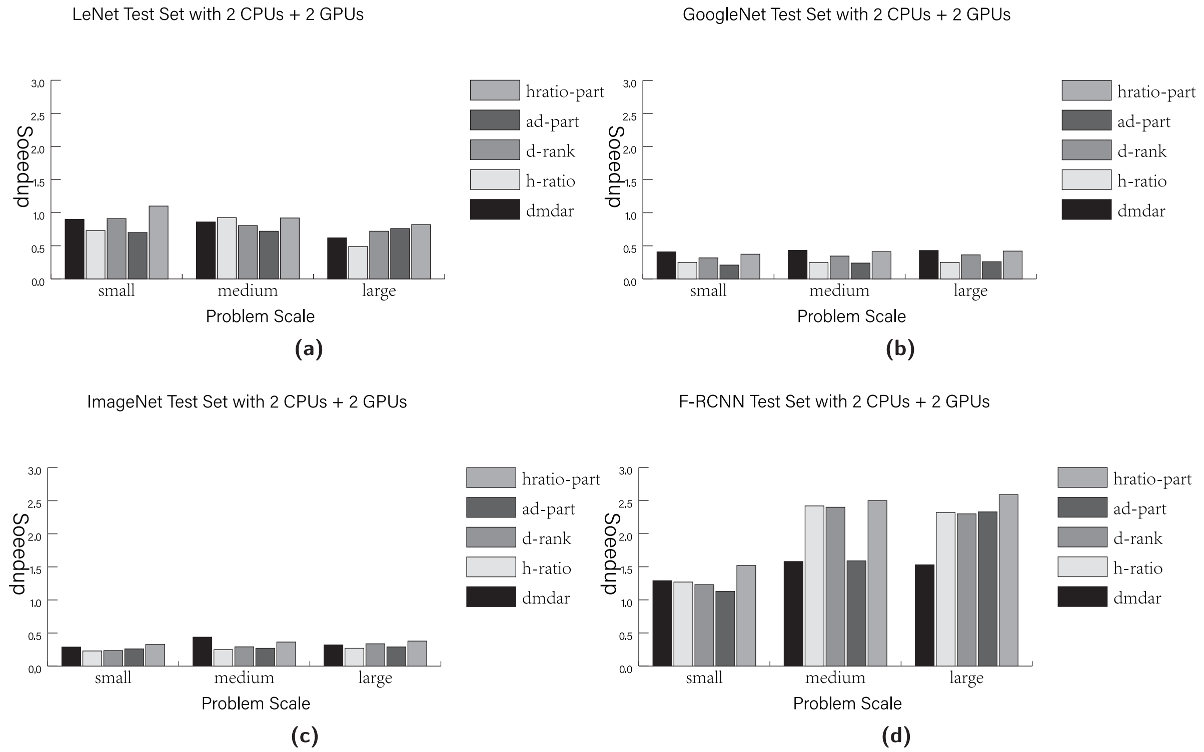

Fig. 8. Experimental results of NN simulation (extended benchmarks in Table 6) on 2CPUs+2GPUs with the configuration in Table 5.

operations can be largely improved by being implemented directly on the GPU. Therefore, the baseline FIFO-based algorithm can perform well when their associated computation is loaded in the GPU. However, many state-of-the-art neural networks require more operations which are hard to be parallelized on the GPU. For instance, F-RCNN [19, 20] contains an interpretation layer with complex sorting operations, such operations render the task partitioning of the neural network throughout the runtime more heterogeneous, which can be significantly optimized by heterogeneity-based algorithms. Therefore, the "hratio-part" algorithm achieves more advantages over other algorithms on F-RCNN over LeNet, GoogleNet and ImageNet.

6.2.3. Experiments for the impact of the threshold setup

As mentioned in Section 4.1, a threshold is set partially for preventing overflow under performance bottlenecks. To evaluate the efficacy of the threshold setup, we conduct a preliminary study to investigate the frequency of overflow occurrence under the no-threshold setup upon large test scenarios (as in Table 3). Table 7 demonstrates the study results which indicate that the system only incurs quite limited amount of overflows by revoking the threshold setups. In particular, the eager scheduler incurs almost no overflow under multiple test scenarios.

7. Related work

Scheduling algorithms for heterogeneous systems. The general problem of scheduling in heterogeneous systems has received much attention. A number of scheduling heuristics have been proposed for scheduling directed acyclic graph-based (DAG) applications in heterogeneous systems [24–29]. These algorithms schedule a single DAG (Directed Acyclic Graph) of tasks onto heterogeneous processing units with varying speed for minimizing the completion time. Zhao et. [30] proposed multi-DAG scheduling by merging multiple DAGs into one DAG. However, such algorithms do not specifically target the CPU/GPU platform, and thus ignore several critical factors when making scheduling decisions, including non-preemptivity, data transfer cost among CPUs and GPUs, data partitioning. Moreover, these existing algorithms are mostly greedy in nature and do not provide a theoretical understanding of the mapping problem considered herein. Only a limited number of approaches have been tested in real systems, e.g., researchers in [31] explore the multitude of real-time multi-GPU configurations. In recent work, dynamic scheduling for multi-core heterogeneous systems [32–35] is still a interesting topic. Researchers are also concerned about task scheduling in restricted scenarios, for instance, researchers in [36] focus on energy constraints, researchers in [37] focus on equipment constraints.

Runtime system support and execution engines for heterogeneous CPU/GPU processors. CPU/GPU processors have become increasingly adopted in various domains, e.g., software development and testing [38–42], cloud computing [43–45], IoT systems [46–48]. For heterogeneous CPU/GPU platforms, a number of runtime systems have been developed to perform task scheduling. PTask [49] focuses on eliminating performance interference of GPU sharing. TimeGraph [9] and others [50] provides prioritization and isolation capabilities in GPU resource management. Harmony [51] schedules translated CUDA code on various devices. Qilin [3] provides an adaptive mapping to automatically partition tasks on a CPU and a GPU. SKMD [14] transparently translate single OpenCL [2] task into variations and execute them on multiple GPUs simultaneously. The aforementioned runtime systems either focus on single task or did not consider task affinities. Some other runtime systems focus on task dataflow parallelism: OmpSs [52], Hydra [53], StreamIt [54], IDEA [55], Liquid Metal [56], Lime [57]. However, these systems do not focus on scheduling multiple graphs onto heterogeneous processors for minimizing the completion time. The recent research trends are more migrated to high-performance computing (HPC) platforms for solving data parallelism problems. HESSLE-FREE [58] leverages fuzzy control for runtime resource management on heterogeneous systems. PLB-HAC [59] proposes a dynamic load-balancing method and implements it on heterogeneous clusters containing combinations of CPUs and accelerators. Northup [60] applies divide-and-conquer programming in heterogeneous systems.

StarPU runtime system. The StarPU [11] runtime system provides programmers with a portable interface for dynamically mapping tasks onto heterogeneous processors (CPUs and GPUs). It integrates development tuning and sampling with several pre-defined task scheduling strategies [15] as plugins. These include the eager scheduler that uses the minimum-completion-time-first policy [24], the dm scheduler that performs an HEFT-based scheduling policy, and several variations of the dm scheduler. Among all pre-defined schedulers, the best one is the dmdar (deque model data aware ready) scheduler. The dmdar scheduler similar to the dm scheduler, but taking data transfer time into account and sorting tasks on a per-worker queue basis. Sc_hypervisor [61] is an extension based on StarPU, which supports co-execution of multiple applications each using the StarPU runtime system. It focuses on partitioning approaches, which split computing resources into isolated sets, and then apply existing StarPU schedulers on each set. However, the StarPU runtime system does not focus on designing efficient mapping algorithms to minimize the completion time, but rather contributes in providing a portable interface for programmers to easily utilize GPUs. The StarPU pre-defined schedulers are mainly designed to handle the single application scenario and use simplified criterion to make mapping decisions. By far, StarPU has been actively maintained and adopted for various research purposes [62–64].

Neural network acceleration. The recent advances of deep neural networks [65–71] lead to increasing computational complexity and thus generate excessive energy and time consumption, e.g., Fast R-CNN [19,20]. For an embedded real-time system, it is necessary to optimize the prediction accuracy of deep neural networks while ensuring the real-time performance. On this purpose, many approaches adopts trade-offs. In particular, some approaches use a convolutional neural network (CNN) with a relatively low relative prediction accuracy. For instance, SSD [72] provides a single shot multiBox detector which completely eliminates proposal generation and subsequent pixel or feature resampling stages and encapsulates all computation in a single network. YOLO [73], provides a single network implemented in the whole detection pipeline. In addition, some approaches focus on optimizing computation process. For instance, ISAAC [74] provides a convolutional neural network accelerator with in situ analog arithmetic in crossbars for enhancing the efficiency of large number of multiply-accumulate (dot-product) operations. EIE [75] works as an energy efficient inference engine on compressed neural network and accelerates the resulting sparse matrix–vector multiplication with weight sharing. Cambricon-x [76] provides a method to exploit the sparsity and irregularity of neural network models for increasing efficiency. UNPU [77] provides a unified DNN accelerator with fully-variable weight bit-precision. Moreover, some other approaches focus on FPGA design and implementation. Researchers in [78] provides an analytical design scheme to identify the solution with best performance and lowest FPGA resource requirement based on a roofline model. Researchers in [79] provides a dynamic-precision data quantization method and a convolver design to improve the bandwidth and resource utilization for convolutional neural networks.

Task mapping application. The recent advances of task mapping applications are widely used in many different embedded systems, including application workloads in modern MPSoC-based embedded systems [80]. Researchers in [80] provide a hybrid task mapping algorithm that combines a static mapping exploration and a dynamic mapping optimization to achieve an overall improvement of system efficiency. In addition, the task mapping applications also include the field of high-performance computing, e.g., researchers in [81] focus on topology-aware task-mapping methods on supercomputers. TASKWORK [82] provides a cloud-aware runtime system for elastic task-parallel HPC applications.

8. Conclusion

In this paper, we investigate the problem of mapping multiple applications implemented using task graphs in a heterogeneous system consisting of CPUs and GPUs. To achieve fast competition time, we present a fine-grain mapping framework that explores a set of critical factors that are suggested by several measurements-based case studies. We present a theoretical framework that formulates this problem as an integer program and a set of practically efficient mapping algorithms. We implement the proposed algorithms in a real heterogeneous system and conduct extensive experiments using a set of popular benchmarks. Experimental results demonstrate that our proposed algorithms can achieve up to 30% faster completion time compared to the state-of-the-art mapping techniques, and can perform consistently superior across different workloads.

In our extensive work, for strengthening the performance of mapping tasks upon deep-neural-network applications under limited resources, we propose an extensive algorithm, namely *heterogeneity ratio-based and data-partition optimizing scheduling (hratio-part)*, that replaces *EFT* with *PFT* and revoke the threshold setups. To evaluate the efficacy of the extensive algorithm, we also extended the experimental setups for server-level scenarios and the benchmarks for simulating the DNN-based applications under limited resources. The experimental results demonstrate that our proposed algorithms, i.e., h-ratio, d-rank, “hratio-part”, can achieve over 10% faster completion time compared to the dmdar algorithm in multiple test scenarios. The experimental results also reveal that the extensive “hratio-part” algorithm can achieve better advantages over the dmdar algorithm than the “h-ratio” and “d-rank” under both the resource-limited and resourceful scenarios.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Shenzhen Peacock Plan, China (Grant No. KQTD2016112514355531), and Science and Technology Innovation Committee Foundation of Shenzhen, China (Grant No. JCYJ20170817110848086). This work is also partially supported by the Climbing Project of China under Grant No. pdjh2019c438. We would like to thank Yiwei Cheng for his help on collecting the trace data and Mingyuan Wu for his help on simulations of the GPU version of this work. We also would like to thank anonymous reviewers and editors whose comments helped us to improve the paper.

References

[1] C. Nvidia, Compute unified device architecture programming guide, 2014, https://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[2] Khronos Corporation, The OpenCL Language, 2011, www.khronos.org/opencl.

[3] C. Luk, S. Hong, H. Kim, Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, in: Proc. of the 42nd International Symp. on Microarchitecture, 2009, pp. 45–55.

[4] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, S. Thibault, Starpu-mpi: Task programming over clusters of machines enhanced with accelerators, in: Recent Advances in the Message Passing Interface, Springer, 2012, pp. 298–299.

[5] U. Dastgeer, C. Kessler, S. Thibault, et al., Flexible runtime support for efficient skeleton programming on hybrid systems, in: International Conference on Parallel Computing (ParCo), pp. 1–8.

[6] A. Bhatele, L.V. Kale, Application-specific topology-aware mapping for three dimensional topologies, in: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, IEEE, 2008, pp. 1–8.

[7] J. Enmyren, C.W. Kessler, Skepu: a multi-backend skeleton programming library for multi-gpu systems, in: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, ACM, 2010, pp. 5–14.

[8] D. Grewe, M.F. O'Boyle, A static task partitioning approach for heterogeneous systems using opencl, in: Compiler Construction, Springer, 2011, pp. 286–305.

[9] S. Kato, K. Lakshmanan, R. Rajkumar, Y. Ishikawa, Timegraph: Gpu scheduling for real-time multi-tasking environments, in: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, in: USENIXATC'11, USENIX Association, Berkeley, CA, USA, 2011, pp. 17–30.

[10] H. Zhou, C. Liu, Task mapping in heterogeneous embedded systems for fast completion time, in: 2014 International Conference on Embedded Software (EMSOFT), IEEE, 2014, pp. 1–10.

[11] R.N.C. Augonnet, Samuel Thibault, P. Wacrenier, Starpu: A unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput. : Pract. Exper. 23 (2) (2011) 187–198.

[12] G. Elliott, J.H. Anderson, Real-world constraints of GPUs in real-time systems, in: Proceedings of the First International Workshop on Cyber-Physical Systems, Networks, and Applications, 2011, pp. 48–54.

[13] C. Basaran, K.-D. Kang, Supporting preemptive task executions and memory copies in gpgpus, in: Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on, IEEE, 2012, pp. 287–296.

[14] J. Lee, M. Samadi, Y. Park, S. Mahlke, Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems, in: Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, in: PACT '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 245–256.

[15] National institute for research in computer science and control, How to optimize performance with starpu, 2008, http://runtime.bordeaux.inria.fr/StarPU/doc/html/HowToOptimizePerformanceWithStarPU.html.

[16] E.T. Grochowski, M.D. Upton, G.Z. Chrysos, C.C. Zhang, M.L. Al-Aqrabawi, Generational Thread Scheduler Using Reservations for Fair Scheduling, Google Patents, 2016, pp. 1–18, US Patent 9, 465, 670.

[17] National institute for research in computer science and control, Starpu-dmdar, 2020, http://starpu.gforge.inria.fr/doc/html/Scheduling.html.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.

[19] R. Girshick, Fast r-cnn, in: The IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1440–1448.

[20] S. Ren, K. He, R. Girshick, J. Sun, Faster r-cnn: Towards real-time object detection with region proposal networks, in: Advances in Neural Information Processing Systems, 2015, pp. 91–99.

[21] R. Al-Jawfi, Handwriting arabic character recognition lenet using neural network, Int. Arab J. Inf. Technol. 6 (3) (2009) 304–309.

[22] P. Ballester, R.M. Araujo, On the performance of GoogLeNet and AlexNet applied to sketches, in: Thirtieth AAAI Conference on Artificial Intelligence, pp. 1124–1128.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, Ieee, 2009, pp. 248–255.

[24] H. Topcuouglu, S. Hariri, M.-y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.

[25] L.F. Bittencourt, R. Sakellariou, E.R. Madeira, Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm, in: Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, IEEE, 2010, pp. 27–34.

[26] H. Zhao, R. Sakellariou, An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm, in: Euro-Par, 2003, pp. 189–194.

[27] H. Arabnejad, J.G. Barbosa, List scheduling algorithm for heterogeneous systems by an optimistic cost table, IEEE Trans. Parallel Distrib. Syst. 25 (3) (2014) 682–694.

[28] R. Sakellariou, H. Zhao, A hybrid heuristic for dag scheduling on heterogeneous systems, in: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, IEEE, 2004, p. 111.

[29] L.-C. Canon, E. Jeannot, R. Sakellariou, W. Zheng, Comparative evaluation of the robustness of dag scheduling heuristics, in: Grid Computing, Springer, 2008, pp. 73–84.

[30] H. Zhao, R. Sakellariou, Scheduling multiple dags onto heterogeneous systems, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, IEEE, 2006, pp. 14–pp.

[31] G.A. Elliott, J.H. Anderson, Exploring the multitude of real-time multi-gpu configurations, in: 2014 IEEE Real-Time Systems Symposium, IEEE, 2014, pp. 260–271.

[32] S. Paul, N. Chatterjee, P. Ghosal, Dynamic task mapping and scheduling with temperature-awareness on network-on-chip based multicore systems, J. Syst. Archit. 98 (2019) 271–288.

[33] T. Yang, Q. Deng, L. Sun, Building real-time parallel task systems on multi-cores: A hierarchical scheduling approach, J. Syst. Archit. 92 (2019) 1–11.

[34] Y. Sun, M. Di Natale, Pessimism in multicore global schedulability analysis, J. Syst. Archit. 97 (2019) 142–152.

[35] N. Chatterjee, S. Paul, S. Chattopadhyay, Task mapping and scheduling for network-on-chip based multi-core platform with transient faults, J. Syst. Archit. 83 (2018) 34–56.

[36] J. Zhou, J. Yan, K. Cao, Y. Tan, T. Wei, M. Chen, G. Zhang, X. Chen, S. Hu, Thermal-aware correlated two-level scheduling of real-time tasks with reduced processor energy on heterogeneous mpsocs, J. Syst. Archit. 82 (2018) 1–11.

[37] C. Hartmann, U. Margull, Gpuart-an application-based limited preemptive gpu real-time scheduler for embedded systems, J. Syst. Archit. 97 (2019) 304–319.

[38] J. Hua, Y. Zhang, Y. Zhang, S. Khurshid, Edsketch: execution-driven sketching for java, STTT 21 (3) (2019) 249–265.

[39] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, S. Khurshid, An empirical study of boosting spectrum-based fault localization via pagerank, IEEE Trans. Softw. Eng. (2019) 1.

[40] M. Wu, H. Zhou, L. Zhang, C. Liu, Y. Zhang, Characterizing and detecting CUDA program bugs, 2019, arXiv preprint:1905.01833.

[41] M. Wu, Y. Ouyang, H. Zhou, L. Zhang, C. Liu, Y. Zhang, Simulee: Detecting cuda synchronization bugs via memory-access modeling, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, in: ICSE '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 937–948.

[42] M. Wu, L. Zhang, C. Liu, S.H. Tan, Y. Zhang, Automating cuda synchronization via program transformation, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 748–759.

[43] D. Yu, Y. Jin, Y. Zhang, X. Zheng, A survey on security issues in services communication of microservices-enabled fog applications, Concurr. Comput. Pract. Exp. 31 (22) (2019).

[44] T. Zheng, Y. Zhang, X. Zheng, M. Fu, X. Liu, Bigvm: A multi-layer-microservice-based platform for deploying saas, in: Fifth International Conference on Advanced Cloud and Big Data, CBD 2017, Shanghai, China, August 13-16, 2017, IEEE Computer Society, 2017, pp. 45–50.

[45] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, X. Liu, Smartvm: a sla-aware microservice deployment framework, World Wide Web 22 (1) (2019) 275–293.

[46] W. Yánez, R. Mahmud, R. Bahsoon, Y. Zhang, R. Buyya, Data allocation mechanism for internet of things systems with blockchain, IEEE Internet Things J. (2020) 1.

[47] I. Yen, S. Zhang, F. Bastani, Y. Zhang, A framework for iot-based monitoring and diagnosis of manufacturing systems, in: 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2017, pp. 1–8.

[48] I. I-Ling Yen, F. Bastani, W. Zhu, H. Moeini, S. Hwang, Y. Zhang, Service-oriented iot modeling and its deviation from software services, in: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018, pp. 40–47.

[49] C.J. Rossbach, J. Currey, M. Silberstein, B. Ray, E. Witchel, Ptask: Operating system abstractions to manage gpus as compute devices, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, in: SOSP '11, ACM, New York, NY, USA, 2011, pp. 233–248.

[50] U. Verner, A. Schuster, M. Silberstein, Processing data streams with hard real-time constraints on heterogeneous systems, in: Proceedings of the International Conference on Supercomputing, ACM, 2011, pp. 120–129.

[51] G.F. Diamos, S. Yalamanchili, Harmony: An execution model and runtime for heterogeneous many core systems, in: Proceedings of the 17th International Symposium on High Performance Distributed Computing, in: HPDC '08, ACM, New York, NY, USA, 2008, pp. 197–200.

[52] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R.M. Badia, E. Ayguade, J. Labarta, Productive cluster programming with ompss, in: Proceedings of the 17th International Conference on Parallel Processing - Volume Part I, in: Euro-Par'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 555–566.

[53] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, P. Wyckoff, Tapping into the fountain of cpus: on operating system support for programmable devices, Oper. Syst. Rev. 42 (2) (2008) 179–188.

[54] W. Thies, M. Karczmarek, S.P. Amarasinghe, Streamit: A language for streaming applications, in: Proceedings of the 11th International Conference on Compiler Construction, in: CC '02, Springer-Verlag, London, UK, UK, 2002, pp. 179–196.

[55] J. Currey, S. Baker, C. Rossbach, Supporting iteration in a heterogeneous dataflow engine, 1–6.

[56] S.S. Huang, A. Hormati, D.F. Bacon, R. Rabbah, Liquid metal: Object-oriented programming across the hardware/software boundary, in: Proceedings of the 22Nd European Conference on Object-Oriented Programming, in: ECOOP '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 76–103.

[57] J. Auerbach, D.F. Bacon, P. Cheng, R. Rabbah, Lime: A java-compatible and synthesizable language for heterogeneous architectures, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, in: OOPSLA '10, ACM, New York, NY, USA, 2010, pp. 89–108.

[58] K. Moazzemi, B. Maity, S. Yi, A.M. Rahmani, N. Dutt, Hessle-free: Heterogeneous systems leveraging fuzzy control for runtime resource management, ACM Trans. Embedded Comput. Syst. (TECS) 18 (5s) (2019) 1–19.

[59] L. Sant'Ana, D. Cordeiro, R.Y. de Camargo, Plb-hac: Dynamic load-balancing for heterogeneous accelerator clusters, in: European Conference on Parallel Processing, Springer, 2019, pp. 197–209.

[60] S. Che, J. Yin, Northup: Divide-and-conquer programming in systems with heterogeneous memories and processors, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2019, pp. 335–344.

[61] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, R. Namyst, Composing multiple starpu applications over heterogeneous machines: A supervised approach, in: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, in: IPDPSW '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1050–1059.

[62] National institute for research in computer science and control, Starpu, 2020, http://starpu.gforge.inria.fr/.

[63] L.L. Nesi, L.M. Schnorr, Investigating memory operations performance in the starpu runtime, 2018, https://www.inf.ufrgs.br/gppd/wsppd/2018/slides/WSPPD_2018_slides_7.pdf.

[64] M.C. Miletto, L. Schnorr, Openmp and starpu abreast: the impact of runtime in task-based block qr factorization performance, in: Anais Do XX Simpósio Em Sistemas Computacionais de Alto Desempenho, SBC, 2019, pp. 25–36.

[65] Y. Cheng, D. Wang, P. Zhou, T. Zhang, A survey of model compression and acceleration for deep neural networks, 2017, arXiv preprint arXiv:1710.09282.

[66] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, E.S. Chung, Accelerating deep convolutional neural networks using specialized hardware, Microsoft Res. Whitepaper 2 (11) (2015) 1–4.

[67] M. Zhang, Y. Zhang, L. Zhang, C. Liu, S. Khurshid, DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, 2018, pp. 132–142.

[68] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, C. Liu, DeepBillboard: Systematic physical-world testing of autonomous driving systems, in: Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, Korea, May 23 - 29, 2020, 2020, pp. 347–358.

[69] X. Li, W. Li, Y. Zhang, L. Zhang, DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019, 2019, pp. 169–180.

[70] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, G. Xu, Reinforcement-learning-guided source code summarization via hierarchical attention, IEEE Trans. Softw. Eng. (2020) 1.

[71] W. Wang, Y. Zhang, Z. Zeng, G. Xu, Trans∧3: A transformer-based framework for unifying code summarization and code search, 2020, arXiv preprint arXiv:2003.03238.

[72] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A.C. Berg, Ssd: Single shot multibox detector, in: European Conference on Computer Vision, Springer, 2016, pp. 21–37.

[73] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You only look once: Unified, real-time object detection, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 779–788.

[74] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J.P. Strachan, M. Hu, R.S. Williams, V. Srikumar, Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars, ACM SIGARCH Comput. Archit. News 44 (3) (2016) 14–26.

[75] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M.A. Horowitz, W.J. Dally, Eie: efficient inference engine on compressed deep neural network, in: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), IEEE, 2016, pp. 243–254.

[76] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, Y. Chen, Cambricon-x: An accelerator for sparse neural networks, in: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Press, 2016, p. 20.

[77] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, H.-J. Yoo, Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision, in: 2018 IEEE International Solid-State Circuits Conference-(ISSCC), IEEE, 2018, pp. 218–220.

[78] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong, Optimizing fpga-based accelerator design for deep convolutional neural networks, in: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2015, pp. 161–170.

[79] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, et al., Going deeper with embedded fpga platform for convolutional neural network, in: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2016, pp. 26–35.

[80] W. Quan, A.D. Pimentel, A hybrid task mapping algorithm for heterogeneous mpsocs, ACM Trans. Embedded Comput. Syst. (TECS) 14 (1) (2015) 14.

[81] M. Deveci, K. Kaya, B. Uçar, Ü.V. Çatalyürek, Fast and high quality topology-aware task mapping, in: 2015 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2015, pp. 197–206.

[82] S. Kehrer, W. Blochinger, Taskwork: a cloud-aware runtime system for elastic task-parallel hpc applications, in: Proceedings of the 9th International Conference on Cloud Computing and Services Science, SciTePress, 2019, pp. 198–209.

Zexin Li received the B.S. degree from the Southern University of Science and Technology, Shenzhen, China, in 2020. He is currently a Ph.D. student in the University of Texas at Dallas. His research interests focus on real-time and embedded systems.

Yuqun Zhang received the B.S. degree from Tianjin University, Tianjin, China, the M.S. degree from the University of Rochester, Rochester, NY, USA, and the Ph.D. degree from the University of Texas at Austin, Austin, TX, USA. He is now an Assistant Professor with the Southern University of Science and Technology, Shenzhen, China. His research interests include software engineering and services computing.

Ao Ding received the B.S. degree from the Southern University of Science and Technology, Shenzhen, China, in 2020. He is currently a M.S. student in the Southern University of Science and Technology, Shenzhen, China. His research interests include software engineering and computer vision.

Husheng Zhou received the Ph.D. degree in computer science from the University of Texas at Dallas, in 2018. He is currently working in VMware, Austin, Texas, as a member of technical staff. His research interests include real-time systems, autonomous embedded systems and computer security.

Cong Liu received the Ph.D. degree in computer science from the University of North Carolina at Chapel Hill, in Jul. 2013. He is an associate professor in the Department of Computer Science, the University of Texas at Dallas. His research interests include real-time systems and GPGPU. He has published more than 30 papers in premier conferences and journals. He received the Best Paper Award at the 30th IEEE RTSS and the 17th RTCSA. He is a member of the IEEE.