Real-Time Scheduling upon a Host-Centric Acceleration Architecture with Data Offloading

Jinghao Sun[1], Jing Li[2], Zhishan Guo[3], An Zou[4], Xuan Zhang[4], Kunal Agrawal[4], Sanjoy Baruah[4*]
[1]Dalian University of Technology, China; [3]University of Central Florida;
[2]New Jersey Institute of Technology; [4]Washington University, St. Louis;

***Abstract*—Challenging scheduling problems arise in the implementation of cyber-physical systems upon heterogeneous platforms with (serial) data offloading and (parallel) computation. In this paper, we adapt techniques from scheduling theory to model, analyze, and derive scheduling algorithms for real-time workloads on such platforms. We characterize the performance of the proposed algorithms, both analytically via the approximation ratio metric and experimentally through simulation experiments upon synthetic workloads that are justified via a case study on a CPU-GPU platform. The evaluation exposes some divergence between the analytical characterization and experimental one; recommendations that seek to balance such divergent characterizations are made regarding the choice of algorithmic approaches.**

I. Introduction

Many emerging safety-critical cyber-physical systems (CPS), including self-driving cars [1], unmanned aerial vehicles [2], and robotics [3], are characterized by an increased degree of autonomous decision-making, which often relies upon artificial intelligence and machine learning components to perform important operations. Since these components have high data parallelism and are computationally intensive, there is a trend in moving towards parallel and heterogeneous computing architectures in the cyber-physical systems (CPS) domain to cope with the increasing performance requirements. These architectures integrate general-purpose CPUs (known as the *host*) with low-power and high-performance *accelerator* devices, e.g., DSP fabrics, GPUs or FPGAs. Example architectures include the NVIDIA Tegra X1 [4], TI Keystone II [5], and Xilinx UltraScale [6].

This trend makes it imperative that we better understand how to schedule workloads upon the host+accelerator platforms in a manner that is cognizant of real-time constraints and concerns. To fully exploit the performance capabilities of the accelerators, programs written on such platforms often form a host-centric acceleration model that consists of two stages: (i) the host first offloads the code and data to accelerator devices; (ii) the offloaded workloads are then executed upon the computing accelerators.

In this paper, we aim to better understand how to make the host-centric acceleration model work efficiently for real-time and latency-sensitive systems. We report upon our experiences in building and analyzing a framework for scheduling a particular form of real-time workload upon a specific host-centric acceleration architecture. Our motivation and objective in sharing our experiences are to contribute towards a better theoretical understanding of the use of host+accelerator computing platforms in safety-critical CPSs. Specific contributions include the following.

- We propose a formal scheduling-theoretic representation of the scheduling problem upon the host-centric acceleration architectures. Although this model is by no means being proposed as one that is appropriate for representing all real-time workloads upon host+accelerator platforms, we believe that it is suitable and can be easily extended for a reasonably large class of applications. Additionally, we hope that some of the advanced scheduling-theoretic ideas that we have adapted and applied here (borrowed from prior work on flow-shop scheduling [7], two-dimensional bin-packing [8], and parallel job scheduling [9], [10]) can find their applicability in the formalization and solution of other resource allocation and scheduling problems on host+accelerator platforms.
- We develop, prove correct, and characterize the effectiveness of two different algorithms with provable performance properties for solving our scheduling problem on the host-centric acceleration architecture. One of these algorithms is developed from first principles, and the other is derived by leveraging off relationships between our problem and previously studied problems in "traditional" scheduling theory. We believe that both algorithms are relatively simple to understand and practical to implement.
- We supplement our analytical characterization of our algorithms and a comparison of their performance from a theoretical perspective, with an experimental evaluation via extensive simulation on synthetic workloads. The simulation experiments have yielded several interesting insights that may further guide the choice of algorithms for scheduling workloads upon host+accelerator platforms.
- Finally, we demonstrate the practicality of our scheduling-theoretic representation of the host+accelerator scheduling problem and the proposed algorithms via a case study conducted on a real CPU+GPU platform.

II. Problem Specification and Prior Results

We propose a formal model for the scheduling problem upon a host+accelerator platform to meet a particular kind of real-time constraint. More specifically, our model explicitly captures the data offloading between the host and the accelerator and represents the computing workload on the accelerator

* Corresponding author: Sanjoy Baruah, baruah@wustl.edu

2642-7346/20/$31.00 ©2020 IEEE
DOI 10.1109/RTAS48715.2020.00-17

as parallelizable jobs on multiple computing units of the accelerator. Our scheduling-theoretic representation is designed to abstract out the main functionalities of host+accelerator platforms, thereby enabling the derivation of theoretical analyses and scheduling algorithms. In Section VI, we provide a case study to demonstrate that our model is capable of representing the main characteristics of a CPU+GPU platform.

A. Problem Specification

System model. We consider a host-centric acceleration architecture with a host and an accelerator device connected via a non-preemptive bus. The accelerator device consists of multiple computing units (called *processors* throughout this paper). We let m denote the number of processors that are available in the accelerator device.

Workload model. We seek to schedule a collection of n tasks $\mathcal{T} = \{\tau_1, \cdots \tau_n\}$ upon the host-centric acceleration architecture. All the jobs of $\mathcal{T}$ are generated upon the host, and their data must first be serially offloaded to the accelerator device where they may be executed in parallel.

More precisely, each task τ_i of $\mathcal{T}$, consisting of data offloading and (parallel) execution of its job, is characterized by the following parameters:

- x_i is the time duration required for task τ_i to offload data from the host to the accelerator device. The data offloading of each task τ_i is non-preemptive. After the data offloading, the system can start to execute the corresponding (parallel) job of τ_i on the accelerator device. Let $X \overset{\text{def}}{=} \sum_i x_i$ denote the total duration of the data offloading associated with all jobs of task set $\mathcal{T}$.
- $\chi_i \leq m$ is the maximum degree of parallelism of task τ_i, i.e., there is no benefit to assign more than χ_i processors to its job. The computing job of task τ_i is *moldable*, meaning that the number of assigned processors m_i may be any number ($\leq \chi_i$); however, it cannot change once the job starts executing[1].
- A work function $w_i(k)$ is specified to give the total execution requirement of τ_i when assigned $k \in 1, 2, ..., \chi_i$ processors. We ***assume*** that (i) total workloads are *monotone*, i.e., $w_i(k)$ is non-decreasing (this assumption is reasonable, since total workload cannot decrease just by assigning more processors); and (ii) the execution lengths are *monotone*; i.e., $w_i(k)/k$ is non-increasing — by assigning more processors, per-processor execution time will not increase[2]. In particular, we say a job has a *linear speedup* if it has a constant work function, i.e., $w_i(k) = w_i(1)$.
- Let $W_i \overset{\text{def}}{=} w_i(\chi_i) = y_i\chi_i$ denote the total execution requirement of τ_i when assigned the maximum possible number (χ_i) of processors, where y_i denotes the duration within which job τ_i guarantees to complete if executing at maximum parallelism. $W_{\max} \overset{\text{def}}{=} \max_i\{W_i\}$ denotes the largest execution requirement of any job, while $y_{\max} \overset{\text{def}}{=} \max_i\{y_i\}$ denotes the longest execution length of any job, when given the maximum number of processors this job can use. Additionally, we use $W_\Sigma \overset{\text{def}}{=} \sum_i W_i$ to denote the total amount of execution that is needed for the whole system. Note that $y_{\max}$ and $W_{\max}$ may be defined by different jobs: it is not necessary that there be a single job τ_i for which $y_i = y_{\max}$ and $W_i = W_{\max}$.

Problem/Objective. In this paper, we aim to efficiently schedule jobs of $\mathcal{T}$ on the host+accelerator platforms such that the completion time of all jobs (equivalently, the overall duration of the schedule of $\mathcal{T}$, or the *makespan* of $\mathcal{T}$) is minimized. (This objective is closely related to the so-called frame-based real-time task systems where tasks have the same implicit deadline and release their jobs synchronously.)

Worst-case approximation ratio. As our objective is to minimize the overall duration of the schedule, we quantify the effectiveness of an algorithm A by its worst-case approximation ratio: the maximum, over all problem instances, of the ratio between the duration of the schedule for a particular instance obtained by A to that obtained by an optimal algorithm.

III. Algorithm I

An straightforward strategy to the scheduling problem described in Section II is to schedule the jobs in a *work-conserving* manner: sequentially offload jobs and execute them whenever there are idle processors. Specifically, when there are k idle processors, we assign $\min(\chi_i, k)$ idle processors to an offloaded job τ_i. However, this work-conserving strategy may lead to very bad timing behaviors. The following example shows that under this work-conserving strategy, jobs may be forced to execute *sequentially* in the worst case.

Example 1. *Consider the following two jobs τ_1 and τ_2 with linear speedup, where L is some constant:*

jobs	x_i	W_i	χ_i
τ_1	1	$2(m-1)$	$m-1$
τ_2	1	mL	m

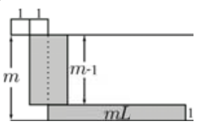

Fig. 1: Work-conserving schedule of jobs in Example 1.

As shown in Figure 1, we sequentially offload the jobs τ_1 and τ_2, and then schedule them by the work-conserving algorithm. At time instant $t = 2$ when τ_2 completes offloading, the only idle processor is assigned to τ_2 according to the work-conserving property. Therefore, the parallel job of τ_2 is enforced to be executed sequentially, and the makespan is $mL + 2$. Clearly, if we do not execute job τ_2 as soon as it completes offloading and, instead, execute τ_2 after the completion of τ_1 (at time 3), τ_2 can be executed in full parallelism. The corresponding makespan is $L + 3$.

The example suggests that it may not be a good strategy to execute a job as soon as it completes offloading. The case becomes even worse when the number of processors m_i

[1]A malleable job, on the contrary, may change the number of processors upon which it runs during its execution, while a *rigid* job needs to execute on a fixed number of processors. The terminology in such classification is not always consistent — we use the terminology as defined in [10].

[2]Note that although some of the theoretical bounds to be derived in this paper requires stricter assumptions (e.g., linear speedup), all algorithms work upon models satisfying the aforementioned (very general) assumptions.

assigned to τ_i is much smaller than the number of processors that the job τ_i needs (χ_i), i.e., $m_i \ll \chi_i$. □

Motivated by this example, we design a strategy to decide the time point at which it is "proper" to assign idle processors to a job. Our strategy, presented in pseudo-code form as Algorithm 1, is characterized by two parameters α and Q, where $\alpha \leq 1$ is a positive real number and Q is an ordering of the jobs to be scheduled. Briefly speaking, Algorithm 1 offloads jobs in the order of Q, and then executes each offloaded job only when there are enough ($\geq \alpha m$) idle processors.

Algorithm 1: GPUSched1(α, Q)

```
while $Q$ is not empty do
    if $|S_{\text{avail}}| \geq \alpha m$ then
        $\tau_i :=$ Dequeue($Q$);
        assign $\min(|S_{\text{avail}}|, \chi_i)$ processors to $\tau_i$
        offload $\tau_i$ to its assigned processors;
        execute $\tau_i$ on its assigned processors;
```

Algorithm 1 operates in the following manner:

1) During run-time, Algorithm 1 maintains, at each instant, the set S_{avail} of processors that are not currently assigned to any job. S_{avail} is initialized to include all m processors; when processors are assigned to a job (Line 4 of Algorithm 1), S_{avail} is updated to remove the assigned processors. Similarly, when a job completes its execution, the processors assigned to it are returned to S_{avail}.
2) Algorithm 1 considers the jobs in $\mathcal{T}$ in the order in which they are present in the queue Q. When considering a job, Algorithm 1 determines whether at least a fraction α of the processors is available. If not, it waits until at least $\alpha\, m$ processors become available following the completion of other jobs to which processors were previously assigned.
3) Once at least $\alpha\, m$ processors are (or become) available, the job τ_i at the head of the queue Q is selected for execution. This entails the following steps:
    a) The maximum number of the available processors that τ_i is able to use, $m_i \leftarrow \min(|S_{\text{avail}}|, \chi_i)$, are assigned to it (and therefore removed from S_{avail}).
    b) τ_i is offloaded from the host to the accelerator device (again in the order in which they appear in Q).
    c) Once offloading is completed, τ_i immediately begins its execution on the assigned m_i processors.

Since the bus connecting the host to the accelerator device is non-preemptive, the data offloading from the host to the accelerator device happens non-preemptively. Hence, multiple jobs may be queued awaiting offloading to the accelerator device via the bus after having been assigned processors by Algorithm 1. However, once a job has completed its offloading phase it immediately begins its execution: there is no additional delay between the completion of the offloading phase and the commencement of execution upon the processors.

A. An Instantiation of Algorithm 1

We now consider an instantiation of Algorithm 1 in which:

- the job with the largest execution requirement (recall from Section II that this execution requirement is denoted $W_{\max}$) is required to be the *last* job in Q; and
- α is assigned a value as follows:

$$\alpha \leftarrow \frac{W_{\max}}{W_{\Sigma}} \tag{1}$$

These values for Q and α have been selected in order to facilitate the derivation of performance bounds for Algorithm 1 that quantify its deviation from optimality (Corollary 1 below). Later in Section III-B we will consider different design choices for Q and α, and will show (Theorem 3) that these choices could result in even better performance.

Theorem 1. *The makespan of any schedule that is generated by the above instantiation of Algorithm 1 for the collection of jobs $\mathcal{T}$ upon m processors is bounded by*

$$\frac{W_{\Sigma}}{m} + X + \max\left(y_{\max}, \frac{W_{\Sigma}}{m}\right) \tag{2}$$

(Recall that $X \overset{\text{def}}{=} \sum_i x_i$ denotes the total data-offloading time; $W_{\Sigma} \overset{\text{def}}{=} \sum_i W_i$ denotes the total execution requirement; and $y_{\max} \overset{\text{def}}{=} \max_i\{W_i/\chi_i\}$ denotes the maximum execution time of job with full parallelism on χ_i processors.)

Proof. Without loss of generality, let us assume that the schedule begins at time-instant zero and ends at time-instant t_f (and hence has a makespan of t_f). As illustrated by Figure 2, we define t_a to be the instant $< t_f$ at which some job that completes at time-instant t_f begins its execution upon the processors that have been assigned to it, and t_b to be the time-instant, also $< t_f$, at which all the data-offloading from the host to the accelerator completes. We note that since a job immediately begins execution when its data has been offloaded from the host to the accelerator, it must be the case that $t_a \leq t_b$. Hence, the makespan t_f can be written as

$$t_f = t_b + (t_f - t_b) \leq t_b + (t_f - t_a) \tag{3}$$

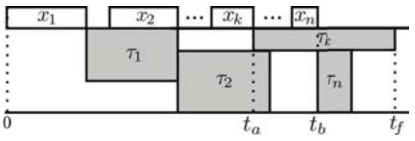

Fig. 2: Illustration for the proof of Theorem. 1

Observe that over the interval $[0, t_b)$, Algorithm 1 does not offload data on the shared bus when it is blocked at line 2 because fewer than αm processors are available — the $m-\alpha m$ processors are actually executing jobs.[3] Let ℓ denote the total duration for which this happens. By the definition of X, we have that

$$t_b = \ell + \sum_i x_i = \ell + X \tag{4}$$

Additionally, we can bound ℓ by the following argument:

[3]Though Algorithm 1 removes processors from S_{avail} when they are assigned to a job instead of when the job begins to execute upon them, it must be the case that these processors are now executing (since there is no data offloading happening, i.e., the data needed for all assigned processors to begin execution has already been offloaded to the accelerator device).

- At each instant in ℓ, more than $(1-\alpha)\times m$ processors are executing. Hence, the total execution that is completed within the intervals of cumulative length ℓ is $> (1-\alpha)m\ell$.
- Since the job with the largest execution requirement, $W_{\max}$, is the last job in queue Q and hence has its data offloaded last, its execution starts at t_b. All the execution done in l must be comprised entirely of the other jobs.

Therefore, we have

$$\begin{aligned} & (1-\alpha)m\ell < (W_\Sigma - W_{\max}) \\ \Leftrightarrow\quad & \ell < \frac{W_\Sigma - W_{\max}}{(1-\alpha)m} \\ \Leftrightarrow\quad & \ell < \frac{W_\Sigma - W_{\max}}{(1-\frac{W_{\max}}{W_\Sigma})m} \quad \text{(By definition of } \alpha\text{: Eqn 1)} \\ \Leftrightarrow\quad & \ell < \frac{W_\Sigma}{m} \end{aligned} \tag{5}$$

From Expressions 4 and 5, we conclude that

$$t_b < \frac{W_\Sigma}{m} + X \tag{6}$$

Consider now the interval $[t_a, t_f)$, during which the last job to complete is executing. Let τ_k denote this job. Consider when τ_k reaches the head of the queue Q in Algorithm 1. There are two possibilities:

- If $\chi_k \leq |S_{\text{avail}}|$ at that point in time, then τ_k is assigned χ_k processors and executes for a duration of W_k/χ_k, which is $\leq y_{\max}$ (by definition of y_{max} — see Section II). In this case, we have
$$(t_f - t_a) \leq y_{\max} \tag{7}$$
- Else, it must be the case that $\chi_k > \alpha m$. Since $|S_{\text{avail}}| \geq \alpha m$ at that point in time, we have $m_k \geq \alpha m$. Therefore, in this case, the execution duration of τ_k is at most $\frac{W_k}{\alpha m}$, i.e.,
$$(t_f - t_a) \leq \frac{W_k}{\alpha m} \leq \frac{W_{\max}}{\alpha m} \tag{8}$$

From Expressions 7 and 8 and by applying the definition of α (in Equation 1), we have

$$(t_f - t_a) \leq \max\left(y_{\max}, \frac{W_{\max}}{\alpha m}\right) \leq \max\left(y_{\max}, \frac{W_\Sigma}{m}\right) \tag{9}$$

Finally from Expressions 3, 6, and 9, we conclude that

$$t_f < \frac{W_\Sigma}{m} + X + \max\left(y_{\max}, \frac{W_\Sigma}{m}\right)$$

and the theorem is proved. □

Corollary 1. *Algorithm 1, with Q and α initialized as described in this section, has a worst-case approximation ratio no larger than* **3** *when scheduling jobs with linear speedup.*

Proof. It is evident that each of the three terms in Expression 2 of Theorem 1 is a lower bound on the makespan of a schedule for the jobs of $\mathcal{T}$ upon m processors:

1) the first term, $\frac{W_\Sigma}{m}$, is a lower bound on the execution duration if all jobs have linear speedup and could execute with full parallelism;
2) the second term X is a lower bound on the data-offloading duration; and
3) in the third term, $\frac{W_\Sigma}{m}$ is a lower bound on the execution duration under full parallelism, and $y_{\max}$ is a lower bound on the duration of a single job with linear speedup.

The Expression 2 is therefore no more than a factor of three larger than the optimal makespan of $\mathcal{T}$ upon m processors. □

B. An Alternative Instantiation of Algorithm 1

In this section, we discuss an alternative instantiation of Algorithm 1 and prove (in Theorem 3) that this alternative instantiation dominates the one discussed in Section III-A: any task set scheduled by the alternative instantiation has a makespan bound no greater than the makespan bound of the original instantiation. Our simulation experiments (in Section V) additionally show that the alternative instantiation tends to have a smaller actual makespan for randomly-generated workloads.

Our alternative instantiation of Algorithm 1 places the job with the largest execution requirement ($W_{\max}$) as the *first* job in Q, and assigns α a value as follows:

$$\alpha \leftarrow \frac{W_{\text{sec}}}{W_\Sigma} \tag{10}$$

where W_{sec} denotes the second-largest execution requirement of jobs in $\mathcal{T}$:

$$\Big(W_{\text{sec}} \leq W_{\max}\Big) \bigwedge \Big(\Big|\{i \mid W_i > W_{\text{sec}}\}\Big| \leq 1\Big) \tag{11}$$

In Theorem 2, below, we will derive a makespan bound for this alternative instantiation of Algorithm 1; in Theorem 3, we will show that this makespan bound is smaller than the one for the original implementaton (which is stated in Theorem 1).

As a notational convenience, in the remainder of this section, we will assume that the jobs are indexed according to the order in which they appear in the queue Q. In other words, we assume that τ_1 has maximum work ($W_1 = W_{\max}$), τ_2 is the second job in Q, τ_3 the third, and so on.

Theorem 2. *With queue ordering and the value of α as specified above, the makespan of any schedule that is generated by Algorithm 1 for the collection of jobs in $\mathcal{T}$ upon m processors is bounded by*

$$\max_{2\leq k\leq n} \{F(k), x_1 + y_1\}$$

where $F(k)$ is defined as follows:

$$F(k) \overset{def}{=} X - \sum_{j=k+1}^{n} x_j + \frac{\left(W_\Sigma - \sum_{j=k}^{n} W_j\right)}{(1-\alpha)\,m} + \max\left(\frac{W_k}{\alpha\, m}, y_k\right) \tag{12}$$

Proof. The main idea of this proof is to show that the RHS of Expression 12 is an upper bound on the makespan: the duration from starting the offloading of the first job to completing the offloading of τ_k plus the duration taken to execute τ_k, when τ_k is the last completed job during the schedule.

Without loss of generality, let us assume that the schedule begins at time-instant zero and ends at time-instant t_f, for a makespan equal to t_f. Let τ_k denote a job that completes at time-instant t_f, and let t_a denote the instant $< t_f$ at which τ_k's data-offloading has completed and it begins execution upon its assigned processors. Note that the first job τ_1 in Q gets to execute with full parallelism on χ_1 processor. If it is the job that completes lastly, the makespan $t_f = x_1 + y_1$.

For the remaining cases of τ_k where $k > 1$, we will show below that the sum of the first three terms in the RHS of Expression 12 is an upper bound on the duration $[0, t_a)$ while the last term is an upper bound on the duration $[t_a, t_f)$.

Consider first the interval $[0, t_a)$. Observe that over this interval, Algorithm 1 will not offload data on the shared bus when it is blocked at line 2 because fewer than αm processors are available. Let ℓ denote the total duration for which this happens. We have that

$$t_a = \left(\sum_{j=1}^{k} x_j\right) + \ell = \left(X - \sum_{j=k+1}^{n} x_j\right) + \ell \tag{13}$$

Additionally, we can bound ℓ by the following argument:

- At each instant in ℓ, more than $(1-\alpha) \times m$ processors are executing; hence, the total execution that is done during intervals in ℓ is greater than $(1-\alpha)m\ell$.
- All this execution must be comprised entirely of the jobs $\tau_1, \tau_2, \ldots, \tau_{k-1}$, as τ_k just finishes its offloading.

Therefore, we have

$$(1-\alpha)m\ell < (\sum_{j=1}^{k-1} W_k) \;\Leftrightarrow\; (1-\alpha)m\ell < (W_\Sigma - \sum_{j=k}^{n} W_j)$$
$$\Leftrightarrow\; \ell < \frac{W_\Sigma - \sum_{j=k}^{n} W_j}{(1-\alpha)m} \tag{14}$$

From Expressions 13 and 14, we conclude that

$$t_a < X - \sum_{j=k+1}^{n} x_j + \frac{\left(W_\Sigma - \sum_{j=k}^{n} W_j\right)}{(1-\alpha)\, m} \tag{15}$$

We now focus on the interval $[t_a, t_f)$. Consider when τ_k, the job that completes at time-instant t_f, reaches the head of the queue Q in Algorithm 1. There are two possibilities:

- If $\chi_k \le |S_{\text{avail}}|$, then τ_k is assigned χ_k processors and executes for a duration $y_k \overset{\text{def}}{=} W_k/\chi_k$ (by definition of y_k — see Section II). In this case, we have
$$(t_f - t_a) \le y_k \tag{16}$$
- Else, $\chi_k > |S_{\text{avail}}|$ and m_k, the number of processors assigned to τ_k, is set equal to $|S_{\text{avail}}|$. Since $|S_{\text{avail}}| \ge \alpha m$, we therefore have $m_k > \alpha m$. Hence, we have
$$(t_f - t_a) < \frac{W_k}{\alpha m} \tag{17}$$

From Expressions 16 and 17, we have

$$(t_f - t_a) \le \max\left(y_k, \frac{W_k}{\alpha m}\right) \tag{18}$$

Summing the bounds on t_a (Expression 15) and $(t_f - t_a)$ (Expression 18), we can bound the makespan t_f by

$$X - \sum_{j=k+1}^{n} x_j + \frac{\left(W_\Sigma - \sum_{j=k}^{n} W_j\right)}{(1-\alpha)\, m} + \max\left(y_k, \frac{W_k}{\alpha\, m}\right)$$

which is the definition of $F(k)$ — see Expression 12. Since we do not necessarily know the identity of τ_k (i.e., which job finishes lastly), we obtain the makespan bound by considering all possibilities and taking the maximum. □

Theorem 3. *The bound of Theorem 2 dominates that of Theorem 1.*

Proof. Suppose that the bound of Theorem 2 achieves its maximum value as $x_1 + y_1$. Recall that the first job τ_1 in Q has $W_1 = W_{\max}$ and is executed with full parallelism on χ_i processor. Hence, the bound of Theorem 2 clearly dominates that of Theorem 1, since

$$x_1 + y_1 \le X + y_{\max} < \frac{W_\Sigma}{m} + X + \max\left(y_{\max}, \frac{W_\Sigma}{m}\right)$$

Suppose that the bound of Theorem 2 achieves its maximum value at k, i.e., $k = \arg\max_{2 \le i \le n}\{F(i)\}$. We consider two cases separately, depending on whether χ_k is larger than $\alpha\, m$.

(i): $\boldsymbol{\chi_k \ge \alpha\, m}$**.** Note that since $y_k \overset{\text{def}}{=} W_k/\chi_k$, it follows from $\chi_k \ge \alpha\, m$ that $y_k \le W_k/(\alpha\, m)$, and the last term (the "max") of Expression 12 equals $W_k/(\alpha\, m)$. We therefore have

$$F(k) \le X - \sum_{j=k+1}^{n} x_j + \frac{\left(W_\Sigma - \sum_{j=k}^{n} W_j\right)}{(1-\alpha)\, m} + \frac{W_k}{\alpha\, m}$$

By algebraic simplification, this can be rewritten as

$$F(k) = X - \sum_{j=k+1}^{n} x_j + \frac{\left(W_\Sigma - \sum_{j=k+1}^{n} W_j\right)}{(1-\alpha)m} + \frac{W_k}{m}\left(\frac{1}{\alpha} - \frac{1}{1-\alpha}\right)$$

Observe that the sum of the first three items in the above expression is maximized when $k = n$. Moreover, since $\alpha \le \frac{1}{2}$, the coefficient of the $\frac{W_k}{m}$ term is non-negative. Furthermore, recall that the job τ_1 has $W_1 = W_{\max}$ while $k > 1$, so the above expression is maximized when $W_k = W_{\text{sec}}$:

$$\begin{aligned}
F(k) &\le X + \frac{W_\Sigma}{(1-\alpha)\, m} + \frac{W_{\text{sec}}}{m}\left(\frac{1}{\alpha} - \frac{1}{1-\alpha}\right) \\
&= X + \frac{W_\Sigma}{(1-\alpha)\, m} + \frac{\alpha\, W_\Sigma}{m}\left(\frac{1}{\alpha} - \frac{1}{1-\alpha}\right) \\
&= X + \frac{W_\Sigma}{m}\left(\frac{1}{1-\alpha} + 1 - \frac{\alpha}{1-\alpha}\right) \\
&= X + \frac{2W_\Sigma}{m}
\end{aligned}$$

which is clearly $\le$ the bound of Theorem 1 (Expression 2).

(ii): $\boldsymbol{\chi_k < \alpha\, m}$**.** Since $y_k \overset{\text{def}}{=} W_k/\chi_k$, it follows that $y_k > W_k/(\alpha\, m)$. and the last term (the "max") of Expression 12 equals y_k. We therefore have

$$F(k) \le X - \sum_{j=k+1}^{n} x_j + \frac{\left(W_\Sigma - \sum_{j=k}^{n} W_j\right)}{(1-\alpha)\, m} + \frac{W_k}{\chi_k}$$

By algebraic simplification, this can be rewritten as

$$F(k) = X - \sum_{j=k+1}^{n} x_j + \frac{\left(W_\Sigma - \sum_{j=k+1}^{n} W_j\right)}{(1-\alpha)\, m} + W_k\left(\frac{1}{\chi_k} - \frac{1}{(1-\alpha)\, m}\right) \tag{19}$$

We now consider two possibilities, depending upon whether the coefficient of W_k in Expression 19 above is ≥ 0.

1) If the coefficient is ≥ 0, then Expression 19 is upper bounded by the case when $k \leftarrow n$ and $W_k \leftarrow W_{\text{sec}}$, and the RHS of Expression 19 is bounded by

$$\begin{aligned} & X + \frac{W_\Sigma}{(1-\alpha)\,m} - \frac{W_{\text{sec}}}{(1-\alpha)\,m} + \frac{W_{\text{sec}}}{\chi_k} \\ & \leq X + \frac{1}{(1-\alpha)\,m}\,(W_\Sigma - W_{\text{sec}}) + y_{\max} \\ & = X + \frac{1}{(1-\alpha)\,m}\,(W_\Sigma - \alpha\,W_\Sigma) + y_{\max} \\ & = X + \frac{W_\Sigma}{m} + y_{\max} \end{aligned}$$

which is clearly $\leq$ the bound of Theorem 1.

2) Otherwise, the coefficient of W_k in Expression 19 is negative, and we may discard this term to obtain the following upper bound on its RHS :

$$\begin{aligned} & X - \sum_{j=k+1}^{n} x_j + \frac{\left(W_\Sigma - \sum_{j=k+1}^{n} W_j\right)}{(1-\alpha)\,m} \\ & \leq X + \frac{W_\Sigma}{(1-\alpha)\,m} \leq X + \frac{2W_\Sigma}{m} \quad (\text{Since } W_{\text{sec}} \leq \tfrac{W_\Sigma}{2}) \end{aligned}$$

which is also obviously $\leq$ the bound of Theorem 1. □

A further improvement. To derive the makespan bound of Theorem 2, the sole constraint placed upon the structure of the queue Q is that the job with the largest workload is at the head of Q. We now specify how the jobs should be ordered in the remainder of the queue to minimize the bound of Theorem 2 (recall that in this section we are indexing the jobs according to their position in Q: τ_1 is at the head of Q, τ_2 is next, and so on.). For each job τ_i in the remainder of the queue ($i = 2, \cdots, n$), we denote its associated weight z_i as follows.

$$z_i = \max\{\frac{W_i}{\alpha\,m}, y_i\} - \frac{W_i}{(1-\alpha)m} \tag{20}$$

We sort the jobs in the reminder of the queue Q in the decreasing order of their weights, i.e., for any $i = 2, \cdots, n-1$, jobs τ_i and $\tau_i + 1$ in Q satisfy $z_i \geq z_{i+1}$. More formally, we use Algorithm 2 to find such an optimal ordering of jobs.

Algorithm 2: Finding an optimal ordering of jobs

```
Input: the set of jobs $\mathcal{T} = \{\tau_1, \cdots \tau_n\}$
Output: a queue $Q$ of the sorted jobs
1 $\tau_1$ := a job with maximum workload (i.e., $W_1 = W_{\max}$)
2 for $i = 1$ up to $n-1$ by $+1$ do
3     $\mathcal{T}' := \mathcal{T}' - \{\tau_i\}$;
4     $\tau_{i+1} := \arg\max_{\tau_i \in \mathcal{T}'}\{z_i\}$
```

Although ordering the queue in this manner does not change the approximation ratio of the algorithm, it does result in improved performance under many (perhaps most) cases. The following lemma explains why this is a good idea.

Lemma 1. *The bound of Theorem 2 achieves its infimum when the queue Q of Algorithm 1 is ordered by Algorithm 2.*

Proof. Let Q^* denote the ordering of the jobs that is determined by Algorithm 2. Suppose, for a contradiction, that some other ordering Q' achieves a smaller bound than the queue Q^*. We let τ'_i denote the i'th job in the queue Q', $\forall i = 1, \cdots, n$, and let z'_j denote the weight of τ'_j as defined in Expression 20. We suppose that j ($j \geq 3$) is the last position where the weight of job τ'_j is larger than the weight of τ'_j's next job, i.e., $z'_j \geq z'_{j+1}$ and $z'_j > z'_{j-1}$ (if such a position j does not exist, it is easy to see that the ordering of the queue Q' must be determined by Algorithm 2, and the lemma trivially follows). We consider the queue Q'' obtained from Q' by:

1) copying the jobs in positions $1, \ldots, (j-2)$ and positions $(j+1), \ldots, n$ of Q' into the same positions in Q'';
2) copying the job in position $j-1$ of queue Q' into position j of queue Q''; and
3) copying the job in position j of Q' into position $j-1$ of queue Q''.

In other words, we obtain Q'' from Q' by "shifting" the j'th and $(j-1)$'th positions of the queue Q', and copying the rest of Q' unchanged.

We observe that there are two positions $j-1$ and j at which the queues Q' and Q'' differ. We will show below that the bound of Theorem 2 is no larger if the ordering of queue Q'' is used, than if the ordering of queue Q' is used. By repeated application of the argument above (equivalently, by induction), it therefore follows that the bound of Theorem 2 is smallest when the ordering of queue Q^* is used.

We now show that the bound of Theorem 2 for the queue Q'' is no larger than it is for Q'. Analogously to τ'_i, let τ''_i denote the job in the i'th position in the queue Q'', $\forall i = 1, 2, \cdots, n$. It follows from the definition of $F(k)$ (Expression 12) that

- For $k \in [1, \ldots, j-2] \cup [j+1, \ldots, n]$, the $F(k)$ values are identical for both queue orderings Q' and Q''. More formally, we use x'_i, y'_i, z'_i and W'_i to represent the parameters of the jobs τ'_i in Q', and use x''_i, y''_i, z''_i and W''_i to represent the parameters of the jobs τ''_i in Q''. The $F(k)$s for the ordering Q' and Q'' are rewritten as $F'(k)$ and $F''(k)$, respectively. From Expression 12, we have

$$\begin{aligned} F'(k) &= \sum_{i=1}^{k} x'_i + \frac{\sum_{i=1}^{k-1} W'_i}{(1-\alpha)m} + \max\{\frac{W'_k}{\alpha m}, y'_k\} \\ &= \sum_{i=1}^{k} x''_i + \frac{\sum_{i=1}^{k-1} W''_i}{(1-\alpha)m} + \max\{\frac{W''_k}{\alpha m}, y''_k\} = F''(k) \end{aligned}$$

- Since $\tau'_j = \tau''_{j-1}$ and $\tau''_j = \tau'_{j-1}$, we have $z''_j = z'_{j-1}$ and $z''_{j-1} = z'_j$. From Expressions 12 and 20, we have

$$\begin{aligned} F'(j) &= \sum_{i=1}^{j} x'_i + \frac{\sum_{i=1}^{j-1} W'_i}{(1-\alpha)m} + \max\{\frac{W'_j}{\alpha m}, y'_j\} \\ &= \sum_{i=1}^{j} x'_i + \frac{\sum_{i=1}^{j} W'_i}{(1-\alpha)m} - \frac{W'_j}{(1-\alpha)m} + \max\{\frac{W'_j}{\alpha m}, y'_j\} \\ &= \sum_{i=1}^{j} x'_i + \frac{\sum_{i=1}^{j} W'_i}{(1-\alpha)m} + z'_j \end{aligned} \tag{21}$$

Since $z'_{j-1} < z'_j$ and $z''_j = z'_{j-1}$, we have $z''_j < z'_j$. Thus,

$$F'(j) > \sum_{i=1}^{j} x'_i + \frac{\sum_{i=1}^{j} W'_i}{(1-\alpha)m} + z''_j$$
$$= \sum_{i=1}^{j} x''_i + \frac{\sum_{i=1}^{j} W''_i}{(1-\alpha)m} + z''_j = F''(j)$$

Moreover, from Expression 21 and $z'_j = z''_{j-1}$, we have

$$F'(j) = \sum_{i=1}^{j} x'_i + \frac{\sum_{i=1}^{j} W'_i}{(1-\alpha)m} + z''_{j-1}$$
$$= \sum_{i=1}^{j} x''_i + \frac{\sum_{i=1}^{j} W''_i}{(1-\alpha)m} + z''_{j-1}$$
$$\geq \sum_{i=1}^{j-1} x''_i + \frac{\sum_{i=1}^{j-1} W''_i}{(1-\alpha)m} + z''_{j-1} = F''(j-1)$$

In sum, we know that $F'(j) \geq F''(j)$ and $F'(j) \geq F''(j-1)$, while for any $k = 1, \cdots, j-2, j+1, \cdots, n$, $F'(k) = F''(k)$. Hence, we have $\max_k F'(k) \geq \max_k F''(k)$, i.e., $\max_k\{F(k)\}$ for the queue ordering Q'' is no larger than that of the queue ordering Q'. The lemma follows. □

IV. Algorithm II

The algorithms in Section III above are derived from first principles. In this section, we propose a different algorithm that builds off prior results in "traditional" scheduling theory. In the following, we first give a brief introduction to the relevant prior work in traditional scheduling theory, exploring the connection between our problem and the relevant traditional scheduling problems.

A. Relevant Prior Research

We now briefly review some prior results from the traditional scheduling theory literature, that we will use in some of the algorithms we derive in this section.

§1. Flow-shop scheduling. In flow-shop scheduling problems, jobs need to be processed upon a number of different machines in a specified order (which is the same for all jobs).

As described in Section II, the host+accelerator platforms can be modeled as a 2-stage flow-shop in which the bus connecting the host to the accelerator is the first stage and the accelerator, the second. There is one bus between the host and the accelerator and multiple identical processors in the accelerator. The problem of scheduling non-parallelizable jobs on such a flow-shop to minimize the overall makespan is referred to as minimizing makespan ($C_{\max}$) in a 2-stage flow-shop ($F2$) in which the first stage has one machine ($P1$) and the second stage has m identical machines (Pm). This problem is known to be NP-hard in the strong sense, even for $m = 2$ (i.e., there are two identical machines in the second stage) [11]. When there is only one machine in each stage, however, *Johnson's rule* [7] solves the problem optimally in polynomial time. We will use Johnson's rule in Section IV; here, we briefly describe the rule.

Johnson's rule: For jobs to be executed upon the 2-stage flow-shop, with the i'th job needing to execute for x_i time on the first stage and y_i time on the second stage. Johnson's rule first partitions the jobs into two disjoint subsets A and B; set A contains all the jobs for which $x_i \leq y_i$ and set B, those for which $x_i > y_i$. All the jobs of set A are scheduled first in non-decreasing order of their x_i parameters, followed by the jobs of set B in non-increasing order of their y_i parameters.

§2. Two-dimensional packing. Given a collection of rectangles and a bin with fixed width and unbounded height, the two-dimensional packing problem [8] seeks to pack the rectangles into the bin such that no two rectangles overlap and the height to which the bin is filled is as small as possible. (The rectangles are assumed oriented: each has a specified side, referred to as its width, that must be parallel to the bottom of the bin.) Coffman et al. [12] proposed an algorithm based on the *first-fit decreasing-height (FFDH)* heuristic for solving this problem. By drawing an analogy between the width of a rectangle and the number of processors needed by a parallel job, it is possible to directly apply the algorithm of [12] to the makespan minimization problem for rigid parallel jobs. We can also apply this algorithm to the makespan minimization problem for *moldable* parallel jobs by simply ignoring the "moldability" of moldable jobs and always assigning exactly χ_i processors to each job τ_i; doing so yields the following result, which we will use later in this paper.

Theorem 4 (From [12, Theorem 3])**.** *Let $\mathcal{T}$ denote a collection of moldable jobs to be scheduled upon m processors. Let $\rho \geq \max_i \{\chi_i/m\}$. The collection of moldable parallel jobs $\mathcal{T}$ can be scheduled upon m processors using the FFDH algorithm of [12] with a makespan no larger than*

$$y_{\max} + \left(\frac{1+\rho}{m}\right) \times W_{\Sigma} \tag{22}$$

§3. Scheduling moldable jobs. Mounie et al. [9] have derived an algorithm with polynomial running time for scheduling monotonic jobs to minimize makespan, and have proved that this algorithm has makespans no larger than $\left(\frac{3}{2}+\epsilon\right)$ times the optimal, for any constant $\epsilon > 0$. Observe that this algorithm can be used to obtain a $\left(\frac{5}{2}+\epsilon\right)$-approximate algorithm for solving our problem, by simply first offloading all the data to the accelerator and then using this $\left(\frac{3}{2}+\epsilon\right)$ algorithm to schedule the computations upon the accelerator.

The algorithm of Mounie et al. [9] is not really "practical": its running time, although indeed polynomial in problem size, depends upon the value of ϵ and tends to become very large for small ϵ. Nevertheless, we have implemented this algorithm, and use it as a baseline for comparison with the other two, more practically implementable, algorithms that we present in Sections III and IV.

B. Instantiation of Algorithm II

We now use algorithms that have previously been proposed for solving the flow-shop scheduling [7] and two-dimensional packing [8], [12] problems (discussed above) as sub-routines to our algorithm. The key idea here is to apply Johnson's algorithm [7] to schedule the jobs with large maximum parallelisms and then use FFDH algorithm [12] to schedule the rest of jobs whose maximum parallelisms are small. The pseudocode of our algorithm is described in Algorithm 3.

Algorithm 3: GPUSched2(ρ)

```
1 Partition $\mathcal{T}$ into $\mathcal{T}_{GE}$ and $\mathcal{T}_{LT}$ (based on $\rho$)
2 Offload the data for
3     jobs in $\mathcal{T}_{GE}$ first, ordered according to Johnson's rule [7]
4     jobs in $\mathcal{T}_{LT}$ next, in arbitrary order
5 Execute
6     jobs in $\mathcal{T}_{GE}$ first, in the order they were offloaded
7     jobs in $\mathcal{T}_{LT}$ next, according to FFDH [12]
```

Algorithm 3 is characterized by a parameter ρ, which is a real number restricted to have a value between zero and one (we will specify later the manner in which its precise value is set). Given this value of ρ, our algorithm (i) *partitions* the jobs into two (disjoint) subsets; (ii) *offloads* the data for the jobs from the host to the accelerator via the shared bus; and (iii) *executes* the jobs on the processors. We now detail how each of these three steps is accomplished.

§i. Partitioning. The jobs in $\mathcal{T}$ are partitioned into two subsets: $\mathcal{T}_{GE}$ has jobs τ_i with $\chi_i \geq \rho m$, and $\mathcal{T}_{LT}$ has jobs τ_i with $\chi_i < \rho m$. [4]

§ii. Offloading. The data for all the jobs in $\mathcal{T}_{GE}$ are offloaded from the host to the accelerator first, followed by the data for all the jobs in $\mathcal{T}_{LT}$. The order in which data-offloading for the jobs in $\mathcal{T}_{GE}$ happens is determined by Johnson's rule [7]:

- All jobs with $x_i \leq y_i$ are scheduled first; amongst these, they are ordered by non-decreasing x_i
- All jobs with $x_i > y_i$ are scheduled next; these are ordered by non-increasing y_i

Data-offloading for the jobs in $\mathcal{T}_{LT}$ is ordered arbitrarily.

§iii. Execution. The jobs are considered for execution in the following order: all the jobs in $\mathcal{T}_{GE}$ are considered first, in the order in which they were offloaded. *Each such job τ_i will be assigned $\underline{\text{exactly}}$ χ_i processors* — the jobs are looked upon as rigid parallel jobs, and their "moldability" is not taken advantage of. A considered job τ_i begins execution when at least χ_i processors become available and executes non-preemptively for a duration of (at most) y_i. The jobs in $\mathcal{T}_{LT}$ are executed once all the jobs in $\mathcal{T}_{GE}$ have completed execution; these jobs are scheduled according to the FFDH [12] algorithm (and the makespan of just these jobs can, therefore, be determined by applying Theorem 4).

Theorem 5. *The makespan of the above schedule for the collection of jobs in $\mathcal{T}$ upon m processors is bounded by*

$$\max\Big(X + \max_{\tau_j \in \mathcal{T}_{GE}}\{x_j\}, \max_{\tau_j \in \mathcal{T}_{GE}}\{y_j\} + \sum_{\tau_j \in \mathcal{T}_{GE}} y_j\Big) + \max_{\tau_j \in \mathcal{T}_{LT}}\{y_j\} + \left(\frac{1+\rho}{m}\right) \times \sum_{\tau_i \in \mathcal{T}_{LT}} W_i \tag{23}$$

[4]The subscripts in $\mathcal{T}_{GE}$ and $\mathcal{T}_{LT}$ denote "Greater than or Equal to" and "Less Than", respectively.

Proof. Without loss of generality, let us assume that the schedule begins at time-instant zero and has a makespan equal to t_f. Let t_a denote the earliest instant by which all the jobs have completed offloading their data from the host to the accelerator, $\underline{\textit{and}}$ all the jobs in $\mathcal{T}_{GE}$ have completed execution on the processors, i.e.,

$$t_a = \max\{X, F_{GE}\} \tag{24}$$

where $X = \sum_{i=1}^{n} x_i$ is the total offloading time, and F_{GE} is the time instant at which all jobs in $\mathcal{T}_{GE}$ complete their execution, as shown in Figure 3. Below we will show that the first line of Expression 23 bounds the duration of the interval $[0, t_a)$, while the second line bounds the duration of the interval $[t_a, t_f)$. The theorem will then follow.

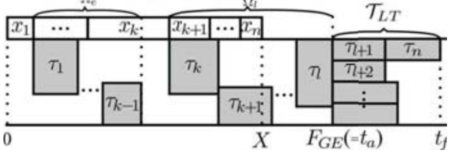

Fig. 3: Illustration for the proof of Theorem. 5

First, consider the interval $[0, t_a)$. By Expression 24, we know that the first line of Expression 23 bounds the duration of interval $[0, t_a)$, if it bounds the completion time F_{GE} of the jobs of $\mathcal{T}_{GE}$. In the following, we derive the upper bound of F_{GE}. As illustrated by Figure 3, let $\tau_k \in \mathcal{T}_{GE}$ denote the last job that had to wait for its data to arrive from the host in order to begin execution on the accelerator. Let Ω_e denote τ_k plus the jobs in $\mathcal{T}_{GE}$ that offloaded their data before τ_k, and $\Omega_l = (\mathcal{T}_{GE} \setminus \Omega_e)$ denote the remaining jobs in $\mathcal{T}_{GE}$. Since τ_k is, by definition, the last job in $\mathcal{T}_{GE}$ that had to wait for data offloading before beginning execution on the accelerator, we can bound the duration of the interval $[0, t_a)$ by

$$F_{GE} \leq \sum_{\tau_j \in \Omega_e} x_j + y_k + \sum_{\tau_j \in \Omega_l} y_j \tag{25}$$

We now consider two possibilities:

1) If $x_k \leq y_k$, it follows from the fact that the jobs in $\mathcal{T}_{GE}$ are ordered according to Johnson's rule that $x_j \leq y_j$ for all $\tau_j \in \Omega_e$. In this case, Inequality 25 yields

$$\begin{aligned} F_{GE} &\leq \textstyle\sum_{\tau_j \in \Omega_e} x_j + y_k + \sum_{\tau_j \in \Omega_l} y_j \\ &\leq \textstyle\sum_{\tau_j \in \Omega_e} y_j + y_k + \sum_{\tau_j \in \Omega_l} y_j \\ &= \textstyle\sum_{\tau_j \in \mathcal{T}_{GE}} y_j + y_k \\ &\leq \textstyle\sum_{\tau_j \in \mathcal{T}_{GE}} y_j + \max_{\tau_j \in \mathcal{T}_{GE}}\{y_j\} \end{aligned} \tag{26}$$

which is the 2nd term in the first line of Expression 23.

2) The other possibility is that $x_k > y_k$, in which case we have that $x_j > y_j$ for all $\tau_j \in \Omega_l$, and Inequality 25 yields

$$\begin{aligned} F_{GE} &\leq \textstyle\sum_{\tau_j \in \Omega_e} x_j + y_k + \sum_{\tau_j \in \Omega_l} y_j \\ &\leq \textstyle\sum_{\tau_j \in \Omega_e} x_j + x_k + \sum_{\tau_j \in \Omega_l} x_j \\ &\leq \textstyle\sum_{\tau_j \in \mathcal{T}_{GE}} x_j + x_k \\ &\leq \textstyle\sum_{\tau_j \in \mathcal{T}_{GE}} x_j + \max_{\tau_j \in \mathcal{T}_{GE}}\{x_j\} \\ &\leq X + \max_{\tau_j \in \mathcal{T}_{GE}}\{x_j\} \end{aligned} \tag{27}$$

which is the first term in the first line of Expression 23. By combining Expressions 26 and 27 into 24, we derive that the first line of Expression 23 bounds the duration of $[0, t_a)$.

Next, consider the duration of the interval $[t_a, t_f)$. Over this interval, our algorithm schedules the jobs in $\mathcal{T}_{LT}$ according to the FFDH algorithm [12]; the makespan of the resulting schedule is, therefore, as specified in Expression 22 of Theorem 4. Note that instantiating Expression 22 upon the collection of jobs in $\mathcal{T}_{LT}$ yields exactly the second line of Expression 23. This completes the proof of Theorem 5. □

In order to complete the specification of the algorithm, it remains to specify a value for the parameter ρ. For this, we use the following result (omitted due to space constraints):

Lemma 2. *Any collection of linear speedup (or non-moldable) jobs in $\mathcal{T}$ that is scheduled by Algorithm 3 with $\rho \leftarrow \left(\sqrt{2}-1\right)$ upon an m processors, has a makespan no larger than*

$$\left(\sqrt{2}+1\right) \times \text{OPT} + \max_{\tau_j \in \mathcal{T}_{LT}} \{y_j\} + \max\left(\max_{\tau_j \in \mathcal{T}_{GE}} \{x_j\}, \max_{\tau_j \in \mathcal{T}_{GE}} \{y_j\}\right) \tag{28}$$

where OPT *denotes the makespan when $\mathcal{T}$ is scheduled upon m processors by an optimal algorithm.*

By observing that each of the "max" terms in Expression 28 is a lower bound on the duration of an optimal schedule, we immediately obtain an approximation ratio of $\left(\sqrt{2}+3\right)$ for Algorithm 3 instantiated with $\rho \leftarrow \left(\sqrt{2}-1\right)$. While this approximation ratio is not particularly good, we observe that Lemma 2 also implies an asymptotic approximation ratio[5] of $\left(\sqrt{2}+1\right)$ for Algorithm 3 instantiated with $\rho \leftarrow \left(\sqrt{2}-1\right)$. This asymptotic approximation ratio compares favorably with the (absolute) approximation ratio of $(\frac{5}{2}+\epsilon)$ that is obtainable (as briefly discussed in Section IV-A.§3) by applying the results of Mounie et al. [9]; hence we have chosen $(\sqrt{2}-1) \approx \mathbf{0.414}$ as the value of ρ in our implementation experiments.

V. Evaluation

In this section, we conduct our comparative evaluation of the algorithms that we discussed above using simulation experiments with randomly-generated synthetic workloads.

Workload generation. Recall that a job τ_i is characterized by a data offloading time x_i, an execution requirement W_i, and the maximum parallelism χ_i. For each simulation run, we randomly generate an instance with n jobs where parameters of each job are generated under uniform distributions unless specified otherwise. Specifically, we first generate a random integer z ranging from $[1, 100]$ for the job size, and a ratio a ranging from $[\frac{\beta}{2}, \beta]$, where $\beta \leq 1$ for the ratio between offloading time and total job size. Then the data-offloading time of a job is az, and the execution requirement is $(1-a)z$. The maximum parallelism of each job ranges uniformly from $[1, m]$, where m is the total number of processors. We construct two kinds of job collections: linear-speedup and non-linear-speedup job collections. In the linear-speedup job collection, each job τ_i has a constant work function, i.e., $w_i(m_i) = W_i$ and $y_i = W_i/\chi_i$, where m_i is the number of processors assigned to τ_i. In the non-linear-speedup job collection, each job τ_i has a work function defined as follows, where $a_i = y_i/2$ is a constant.

$$w_i(m_i) = \begin{cases} a_i m_i & m_i \leq \chi_i/2 \\ 2a_i m_i & m_i > \chi_i/2 \end{cases} \tag{29}$$

Algorithm Comparison. We evaluated four implementations that implements a baseline and our proposed algorithms:

1) ***Impl-I*** implements GPUSched1(α, Q) (listed as Algorithm 1), as initially described in Section III-A with $\alpha \leftarrow W_{\max}/W_{\Sigma}$ and the job with execution requirement $W_{\max}$ at the end of Q.
2) ***Impl-I**** implements GPUSched1(α, Q), as described in Section III-B with $\alpha \leftarrow W_{\text{sec}}/W_{\Sigma}$,[6] the job with execution requirement $W_{\max}$ at the front (head) of Q, and the remainder of Q ordered according to the pseudocode listed as Algorithm 2.
3) ***Impl-II*** implements GPUSched2(ρ), listed as Algorithm 3 in Section IV with $\rho \leftarrow (\sqrt{2}-1)$.
4) ***Impl-B*** (the "B" here stands for "baseline") implements the algorithm in [9] for scheduling moldable jobs discussed in Section IV-A§3. (In this implementation, all the data was first offloaded to the accelerator; after that, the problem of executing the jobs on the processors of the accelerator was solved as scheduling independent moldable jobs.)

We applied an additional optimization to Impl-I and Impl-I* with regards to data offloading. Line 2 of Algorithm 1 states that a job starts its data-offloading only if there are enough (i.e., at least $\alpha\, m$) available processors. Here, we implement the improved algorithms by a bit of look-ahead in the data offloading: when job τ_i is at the head of the queue, although there are not αm available processors at the current time, we still start the offloading for τ_i if we know that there will be αm available processors x_i time units later. (Note that this does not affect the theoretical bounds derived in previous sections.)

Evaluation criterion. One can calculate a lower bound on the optimal makespan for any problem instance by:

$$\text{LB} = \max\{d^* + \min_i\{x_i\}, X + \min_i\{y_i\}\} \tag{30}$$

where d^* is set equal to two-thirds of the duration required by Impl-B on the processors[7]. We quantify the goodness of a schedule for an instance by its makespan divided by LB. Hence, for a given problem instance, an algorithm, for which this ratio is smaller, is "better" at scheduling that instance.

[5]The *asymptotic* approximation ratio of an algorithm compares the algorithm's performance with that of an optimal algorithm as the problem size — in our case, the number of jobs — becomes very large ($\rightarrow \infty$: approaches infinity). As $n \rightarrow \infty$, the two additive terms in Expression 28 become negligibly small in comparison with OPT, and hence do not contribute to the asymptotic approximation ratio.

[6]W_{sec} defined in Equation (11) is the second-largest execution requirement.

[7]Since the algorithm of [9], used by Impl-B for scheduling the jobs on processors, is a $(\frac{3}{2}+\epsilon)$-approximation, it follows that $\frac{3}{2}$ of the duration required by Impl-B on processors is a lower bound on the optimal makespan.

Evaluation Results. We present three sets of results from the experiments with linear-speedup job collections and non-linear-speedup job collection respectively. We use the approximation ratio (over LB) as the performance metric, which provides a comparison across the different algorithms on the workload with fixed parameters. Out of 1,000 generated workloads under each parameter setting, we report the average, 10th percentile, 50th percentile, and 90th percentile of the approximation ratio of each implementation[8] in Fig. 4 to 6.

We can observe that Impl-B generally performs worse than the other three. This is due to the fact that the baseline Impl-B focuses primarily on minimizing the total processing time of jobs, while overlooking the fact that increasing the overlapping period between data-offloading and execution may also reduce the total makespan. Our proposed methods (e.g., Impl-I, Impl-I* and Impl-II) optimize both aspects in an integrated manner and thus outperform the baseline method Impl-B. In addition, Impl-II and Impl-I* outperform Impl-I in general, since these implementations (compared with Impl-I) apply more complicated mechanisms to minimize the makespan. Moreover, for Impl-II and Impl-I*, the reported 10th, 50th, and 90th percentile approximation ratios (in boxes) stay relatively close to the average, indicating that the performance and relationship of the algorithms are relatively robust (i.e., less variable). We also observed that there is a peak (i.e., worse performance) to Impl-I and Impl-I* when the number of jobs is around 1 to 2 times the number of processors. Furthermore, Impl-I* and Impl-II performs better when the jobs have linear speedup, while Impl-I and Impl-B are insensitive to the types of jobs.

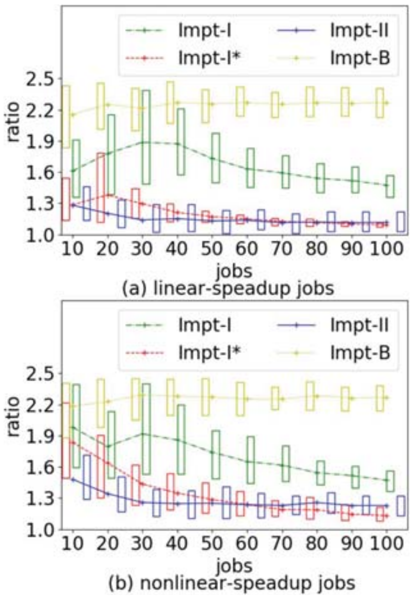

Fig. 4: $m = 16, \beta = 0.5$.

[8]In each set of Fig. 4 to 6, the top of the box represents the 10 percentile, the middle of the box represents the 50 percentile, and the bottom of the box represents the 90 percentile.

Specifically, Fig. 4 reports the implementations' performance on an increasing number of jobs per instance. When the number of jobs becomes larger (than the number of processors), all three approaches, except for Impl-B, tend to perform better. This is as expected since the overlapping between data-offloading and execution upon the processors is more likely to occur when there are more jobs. The larger proportion of overlapping duration indicates the better performance of the approaches. In contrast, the baseline Impl-B does not overlap the data-offloading and execution, so its performance remains the same. Additionally, the performance variation decreases with an increasing number of jobs for all approaches, as the execution of a long job has a better chance to overlap with other jobs when there are more jobs available.

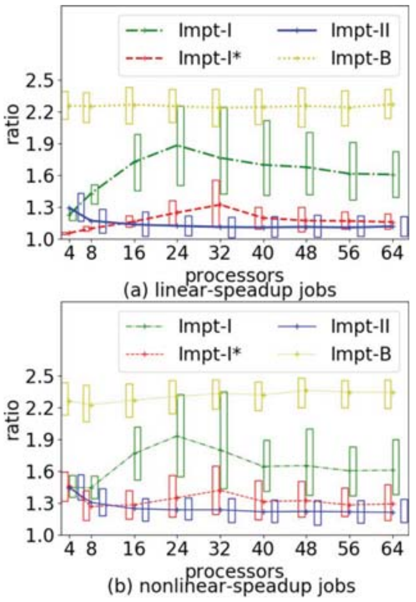

Fig. 5: $n = 50, \beta = 0.5$.

Fig. 5 shows the results when the number of processors increases. The performance of Impl-B stays about the same since (i) it does not allow any overlapping between data offloading and execution, and (ii) the packing problem of the execution part has a stable approximation ratio, leading to a relatively unchanging makespan ratio over the optimal lower bound. The approximation ratio of Impl-II decreases when the number of processors increases, because jobs in $\mathcal{T}_{LT}$ can better utilize the accelerator when there are more processors. Impl-I* outperforms Impl-II only when the number of processors is small, since jobs tend to be executed sequentially under Impl-II with a small number of processors.

Fig. 6 reports the approximation ratio against varying values of β. Recall that a larger β indicates that the execution time is more likely to dominate the total data offloading plus execution duration of jobs. The approximation ratios of all the approaches tend to grow when $\beta < 1/2$, while start to drop soon after β exceeds 1/2. Again, this is not surprising since the computation part dominates when $\beta < 1/2$ — all approaches

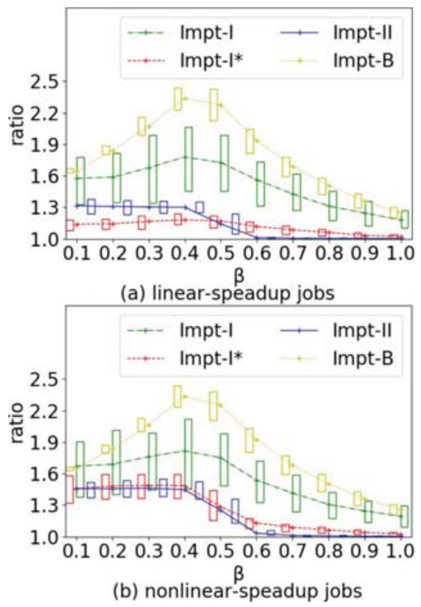

Fig. 6: $n = 50, m = 16$.

suffer from not being able to pack optimally; when β gets larger (than half), the sequential offloading time dominates, and any approach would "pay" the same offloading time, while most executions (in general with shorter lengths) can be easily done in parallel to offloading.

VI. Case Study

In this section, we demonstrate the practicality of our formal scheduling-theoretic representation and the proposed algorithms for the scheduling problem upon the host+accelerator platforms, via a case study using workloads executed on a real CPU+GPU platform. GPUs have been widely used as accelerators for various applications, especially those in autonomous CPSs. For example, Behzad Boroujerdian [2] develops a simulation framework for micro aerial vehicles with CPU+GPU platforms. Many computer vision applications in CPSs, such as object detection for self-driving [1], 3D tracking for humanoid robots [3], and instrument tracking for intra-cardiac surgical procedures [13], must exploit GPUs to accelerate their computations to meet their real-time demands.

Our particular CPU+GPU platform has a host CPU and an NVIDIA TITAN X GPU with 30 streaming multiprocessors (SMs), connected via a single bidirectional peripheral component interconnect express (PCIe) bus. We reserve 2 SMs for system applications and leave 28 SMs dedicated to experimentation. We set the SM and memory frequencies of the GPU at fixed values of 810MHz and 1911MHz, respectively, using the `nvidia-smi` command, and set the GPU to persistence mode to keep the NVIDIA driver loaded even when no applications are accessing the GPU.

For this CPU+GPU platform, SMs are modeled as the computing units upon which the computing workloads can be executed in parallel. In contrast to the extensive works that consider the entire GPU as a single sequential computing unit [14]–[23], in this work, we try to exploit the internal parallelism inside a GPU for efficient execution of concurrent GPU workloads. To the other potential extreme, each SM has smaller execution units that can simultaneously perform the same computation on different data. However, the scheduling inside an SM is controlled by its internal warp scheduler, whose policy is not publicly disclosed. Extensive research efforts have been made to understand the different and complex properties of this internal scheduler on different NVIDIA architecture via extensive experimental testing and validation [24]–[28]. However, for better utilization and power efficiency of the GPU, programmers are not given explicit control to access these smaller execution units in SMs individually. Therefore, in our case study, we choose to model each SM as one computing unit and only allow one SM to execute the computation of a single job at any time, so the scheduler inside an SM does not affect the scheduling between jobs.

We use software interfaces and techniques, including CUDA Stream, persistent threads and SM ID selection [29]–[32], to assign a set of dedicated SMs to a job (according to the processor allocation produced by the proposed algorithms) and allow concurrent execution of multiple GPU jobs. We use `cudaMemcpyAsync` and `cudaStreamSynchronize` to control the data offloading between CPU and GPU.

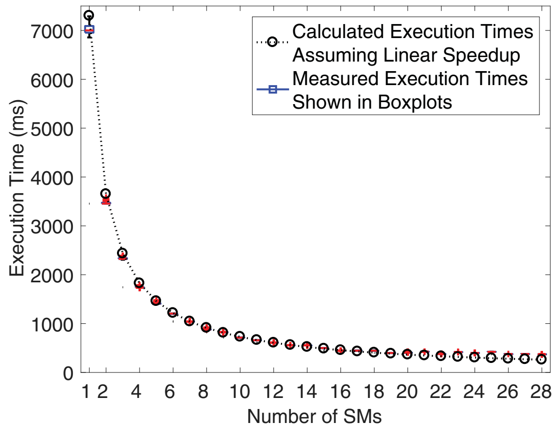

Fig. 7: Execution times of a synthetic GPU computation workload on different number of assigned SMs.

We generate synthetic GPU computation workloads using matrix calculations. Figure 7 shows the execution times of one such workload on varying numbers of assigned SMs. In particular, the boxplots presents the actual execution times for 100 measurements for each setting, where the blue boxes give the first to third quartiles, and the red crosses are the outliers.[9] The dotted curve draws the calculated execution times, assuming that the workload has linear speedup on up to 28 SMs. Comparing the calculated and measured execution times, we can see that the synthetic workload has a linear

[9]As the variance of measured execution times is very small given more than 1 SM, the blue boxes are hidden under the red crosses.

speedup on up to 16 SMs, after which the actual execution times do not decrease further with more SMs assigned. Thus, this synthetic workload has a maximum parallelism degree χ of 16. To obtain the worst-case total work $w(1)$ for the experiment, we conduct the measurement for the workload by occupying the remaining SMs with other busy work.

TABLE I: Makespans (ms) of scaled task sets under different algorithms and different numbers of available SMs

Algo.	Calculated makespan	Measured makespan on diff avail. SMs			
		7 SMs	14 SMs	21 SMs	28 SMs
Impl-I	4200	4088	4156	3997	3950
Impl-I*	3600	3485	3518	3350	3213
Impl-II	3600	3490	3522	3530	3579
Impl-B	4933	4803	4881	4879	4928

For the case study on 28 SMs, we construct a task set with 8 tasks, where the task parameters are shown in Figure 8. We have implemented the different schedules under Impl-I, Impl-I*, Impl-II, and Impl-B for this task set and measured the execution timelines. For example, Figure 8 shows the timeline for Impl-I*. We observe that the measured makespan is smaller than the makespan calculated using the worst-case parameters, mainly because the data offloading and computation take less time in practice than the worst-case values. Also note that task τ_3 has a much shorter computation time, as it is the only job executing in GPU and can make use of all the shared cache and memories. We leave the cache and memory allocation for accelerators as future work. We have also scaled this task set to run on 7, 14, and 21 available SMs, by scaling the total work and assigned SMs of tasks proportionally while keeping the same timelines. The measured makespans on a varying number of available SMs are very similar, as shown in Table I.

VII. Related Work

Heterogeneous host+accelerator computing platforms recently gain a lot of attention. Various techniques have been developed for better utilizing the GPU and speed up the GPU workloads. For example, Liu et al. [33] studied how to execute tree traversal algorithms efficiently on a GPU. It proposed a hybrid inspector-executor approach, where the CPU uses the information of partial execution of the traversals on the GPU to optimize the execution of the remaining portion of the traversals. Lin et al. [34] proposed Integrated Vectorization and Scheduling methods to exploit multiple forms of parallelism for optimizing throughput for synchronous dataflows on memory-constrained CPU-GPU platforms, such as the signal and information processing on embedded systems. For multi-tasking environments in the cloud, persistent threads technique for allocating SMs to programs was proposed [29]–[31]. Wang et al. [35] implemented a user-mode lightweight CPU–GPU resource management framework to optimize the CPU utilization while maintaining good Quality of Service (QoS) of GPU-intensive workloads in the cloud, such as cloud games. This work designed QoS-aware scheduling algorithms for virtual machines running on CPU and GPU via adaptive control. However, all of these works do not consider the data transfer between CPU and GPU.

To demonstrate the importance of analyzing and scheduling data transfer on CPU-GPU platforms, Gregg et al. [36] benchmarked a variety of GPU kernels on different CPU-GPU platforms and showed that data transfer time between CPU and GPU could take from 1x to 50x longer than the GPU processing time. To model data transfer in GPU workloads, [37] create a model to predict the transfer time and estimate the performance of different system mechanisms of overlapping computation and data transfer, such as explicit and implicit memory copy statements and CUDA streams. Chen et al. [38] extend the original Flink on CPU clusters to GFlink on heterogeneous CPU-GPU clusters with multiple CPUs and GPUs for big data. In addition to the new programming framework, device cache and memory management schemes, and locality-aware scheduling scheme, GFlink empirically reduces the communication between JVM and GPU by bulk transfer and asynchronous communication, which enables the communication to overlap with the computation on GPUs. [39] proposed OS abstractions under the dataflow programming model for accelerators such as GPUs to achieve good performance, fairness, and low latency of applications. This abstraction reduces unnecessary data copies and buffering by explicitly specify the origin and destination of the data. In addition to virtualizing a GPU into multiple logical GPUs for better isolation, [40] developed techniques to support data swapping and shared device memory functionality to improve memory-copy throughput. Valery et al. [41] proposed a CPU–GPU collaboration method to speed up Principal Component Analysis and reduce the power consumption on mobile devices, via exploiting the shared memory architecture of System-on-Chips. However, the above works do not consider the real-time requirements of applications.

Much existing work on real-time scheduling on CPU-GPU platforms only considers a GPU as a single sequential computing unit without considering its internal parallelism and concurrent execution of multiple tasks. Specifically, [14] proposed priority-based scheduling policies to address the trade-off between response times and throughput with GPU. [15] presented two methods for integrating a GPU into soft real-time multiprocessor systems, which was extended for multiple GPUs in [16]. [18] optimized the execution of DNN workloads on GPU in a real-time multi-tasking environment. For hard real-time systems, [17] proposed a scheduling approach based on time-division multiplexing in GPU. The problem of thermal-aware and energy-efficient execution on CPU-GPU systems was studied in [21] and [23], while the co-scheduling problem for fused CPU-GPU architectures with shared last level cache was analyzed in [22]. For integrated System-on-Chips, [42] characterized the memory access conflicts. Such conflicts were addressed by a memory-arbitration mechanism proposed in [43] and by a memory bandwidth throttling-based approached developed in [44].

For improving the real-time performance of sequential kernel execution, some works proposed software techniques to

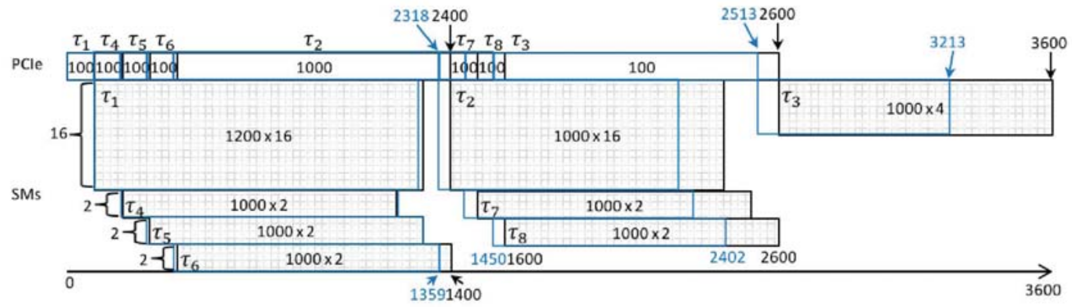

Fig. 8: Calculated execution timeline using the worst-case parameters (black) vs. Measured execution timeline (blue) for 8 tasks scheduled by Impl-I*. τ_2 and τ_3 have 1000ms offloading time, while the other tasks have 100ms offloading time. τ_1, τ_2, and τ_3 have total work of 19200ms, 16000ms, and 4000ms, respectively. τ_4 to τ_8 have a total work of 2000ms. τ_1, τ_2, and τ_3 have maximum parallelism degrees of 16, 16 and 4, respectively, while the others have a maximum parallelism degree of 2.

allow preemption. In particular, [45] supported preemptive data copies via providing preemption points at boundaries between the data chunks, but the GPU kernels are executed sequentially and non-preemptively by fixed-priority scheduling, which was extended via splitting a kernel into sub-kernels with preemption points in user-level [46] and in driver-level [47]. In contrast, [48] transactionized GPU kernels to allow abortion and roll back exploiting the heterogeneous system architecture of System-on-Chips. [19] utilized the new GPU architectural features, such as pixel-level preemption and thread-level preemption, to implement the deadline-based schedulers for GPU on an embedded System on a Chip.

For concurrent scheduling of multiple GPU kernels, there have been some empirical results demonstrating its applicability using memory page coloring and persistent threads [32], and using gang scheduling with fixed preemption points achieved on System-on-Chips [49]. [20] proposed a fine-grained approach for scheduling OpenVX graphs and studied its response time analysis. However, none of the above works provide rigorous theoretical analysis considering the data transfers and parallel execution in a GPU.

VIII. Summary and Perspectives

This paper described our efforts at applying scheduling theory to provide real-time performance to the host+accelerator platforms. We formulate this scheduling problem into a model that was amenable to rigorous schedulability analysis while capturing the main characteristics and abstractions of the platforms. In this work, however, we restricted ourselves to a simplified version of the general model and designed two algorithms: one derived entirely from first principles and hence unique to our problem, the other based upon several very powerful prior results from traditional scheduling theory. We compared both the theoretical and empirical performance of these algorithms and demonstrated the practicality of our model and the proposed algorithms via a case study conducted on a real CPU+GPU platform.

The major limitation of our work is that the workload analyzed in this work is very restricted. In many CPS, we would expect to have recurrent tasks with different deadlines, perhaps with each job (here, an invocation of a task) needing to use the accelerator multiple times (rather than only once), use the bus to transfer back the computation results from the accelerator to the host, and also use the host for part of its computation. Although the model proposed in this work can be easily extended to represent such generalized workloads, it is very challenging to extend the corresponding theoretical analyses and scheduling algorithms. We plan to derive algorithms and analysis tools for scheduling such general workloads on the host+accelerator platforms.

Acknowledgments

This work was supported by the National Science Foundation (CCF-1733873, CCF-1618802, CCF-1439062, CNS-1850851, CNS-1814739, CPS-1932530, CNS-1911460, CNS-1739643, and CNS-1948457) and National Foundation of Science of China (No. 61972076). J. Li and Z. Guo are the co-first authors.

References

[1] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint*, 2017.

[2] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. J. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *Microarchitecture (MICRO), 2018 51th Annual IEEE/ACM International Symposium on*. IEEE, 2018.

[3] P. Michel, J. Chestnutt, S. Kagami, K. Nishiwaki, J. Kuffner, and T. Kanade, "Gpu-accelerated real-time 3d tracking for humanoid locomotion and stair climbing," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE, 2007, pp. 463–469.

[4] T. NVIDIA, "K1: A new era in mobile computing," *Nvidia, Corp., White Paper*, 2014.

[5] T. Instruments, "66ak2hxx multicore keystone ii system-on-chip (soc)," *November*, 2012.

[6] S. Leibson and N. Mehta, "Xilinx ultrascale: The next-generation architecture for your next-generation architecture," *Xilinx White Paper WP435*, vol. 143, 2013.

[7] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 61–68, 1954.

[8] B. S. Baker, E. G. Coffman, Jr, and R. L. Rivest, "Orthogonal packing in two dimensions," *SIAM Journal on Computing*, vol. 9, no. 4, pp. 846–855, 1980.

[9] G. Mounie, C. Rapine, and D. Trystram, “A $\frac{3}{2}$-approximation algorithm for scheduling independent monotonic malleable tasks,” *SIAM Journal on Computing*, vol. 37, no. 2, pp. 401–412, 2007.

[10] K. Jansen, “A (3/2+ϵ) approximation algorithm for scheduling moldable and non-moldable parallel tasks,” in *24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’12, 2012, pp. 224–235.

[11] J. N. D. Gupta, “Two-stage, hybrid flowshop scheduling problem,” *Journal of the Operational Research Society*, vol. 39, no. 4, pp. 359–364, 1988.

[12] E. G. Coffman, Jr, M. R. Garey, D. S. Johnson, and R. E. Tarjan, “Performance bounds for level-oriented two-dimensional packing algorithms,” *SIAM Journal on Computing*, vol. 9, no. 4, pp. 808–826, 1980.

[13] P. M. Novotny, J. A. Stoll, N. V. Vasilyev, J. Pedro, P. E. Dupont, T. E. Zickler, and R. D. Howe, “Gpu based real-time instrument tracking with three-dimensional ultrasound,” *Medical image analysis*, vol. 11, no. 5, pp. 458–464, 2007.

[14] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, “Timegraph: Gpu scheduling for real-time multi-tasking environments,” in *Proc. USENIX ATC*, 2011, pp. 17–30.

[15] G. A. Elliott and J. H. Anderson, “Globally scheduled real-time multiprocessor systems with gpus,” *Real-Time Systems*, vol. 48, no. 1, pp. 34–74, 2012.

[16] G. A. Elliott, B. C. Ward, and J. H. Anderson, “Gpusync: A framework for real-time gpu management,” in *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 2013, pp. 33–44.

[17] V. Golyanik, M. Nasri, and D. Stricker, “Towards scheduling hard real-time image processing tasks on a single gpu,” in *2017 IEEE International Conference on Image Processing (ICIP)*, 2017.

[18] H. Zhou, S. Bateni, and C. Liu, “Sˆ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 190–201.

[19] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, “Deadline-based scheduling for gpu with preemption support,” in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 119–130.

[20] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, “Making openvx really” real time”,” in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 80–93.

[21] S. Hosseinimotlagh and H. Kim, “Thermal-aware servers for real-time tasks on multi-core gpu-integrated embedded systems,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018.

[22] M. Damschen, F. Mueller, and J. Henkel, “Co-scheduling on fused cpu-gpu architectures with shared last level caches,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2337–2347, 2018.

[23] M. H. Santriaji and H. Hoffmann, “Merlot: Architectural support for energy-efficient real-time processing in gpus,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 214–226.

[24] N. Otterness, V. Miller, M. Yang, J. Anderson, F. D. Smith, and S. Wang, “Gpu sharing for image processing in embedded real-time systems,” *OSPERT’16*, 2016.

[25] N. Otterness, M. Yang, T. Amert, J. Anderson, and F. D. Smith, “Inferring the scheduling policies of an embedded cuda gpu,” in *Workshop on Operating Systems Platforms for Embedded Real Time Systems Applications (OSPERT)*, 2017.

[26] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang, “An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads,” in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017, pp. 353–364.

[27] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, “Gpu scheduling on the nvidia tx2: Hidden details revealed,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 104–115.

[28] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, “Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems,” in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[29] C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan, and D. Qian, “Smguard: A flexible and fine-grained resource management framework for gpus,” *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[30] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style gpu programming for gpgpu workloads,” in *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*. IEEE, 2012, pp. 1–14.

[31] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, “Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations,” in *29th ACM on International Conference on Supercomputing*, 2015, pp. 119–130.

[32] S. Jain, I. Baek, S. Wang, and R. R. Rajkumar, “Fractional gpus: Software-based compute and memory bandwidth reservation for gpus,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018.

[33] J. Liu, N. Hegde, and M. Kulkarni, “Hybrid cpu-gpu scheduling and execution of tree traversals,” in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 2.

[34] S. Lin, J. Wu, and S. S. Bhattacharyya, “Memory-constrained vectorization and scheduling of dataflow graphs for hybrid cpu-gpu platforms,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, p. 50, 2018.

[35] B. Wang, R. Ma, Z. Qi, J. Yao, and H. Guan, “A user mode cpu–gpu scheduling framework for hybrid workloads,” *Future Generation Computer Systems*, vol. 63, pp. 25–36, 2016.

[36] C. Gregg and K. Hazelwood, “Where is the data? why you cannot debate cpu vs. gpu performance without the answer,” in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 134–144.

[37] B. Van Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, “Performance models for cpu-gpu data transfers,” in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 11–20.

[38] C. Chen, K. Li, A. Ouyang, Z. Zeng, and K. Li, “Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1275–1288, 2018.

[39] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, “Ptask: operating system abstractions to manage gpus as compute devices,” in *23rd ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 233–248.

[40] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, “Gdev: First-class gpu resource management in the operating system.” in *USENIX Annual Technical Conference*. Boston, MA;, 2012, pp. 401–412.

[41] O. Valery, P. Liu, and J.-J. Wu, “A collaborative cpu–gpu approach for principal component analysis on mobile heterogeneous platforms,” *Journal of Parallel and Distributed Computing*, vol. 120, pp. 44–61, 2018.

[42] R. Cavicchioli, N. Capodieci, and M. Bertogna, “Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms,” in *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017, pp. 1–10.

[43] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, “Sigamma: Server based integrated gpu arbitration mechanism for memory accesses,” in *25th International Conference on Real-Time Networks and Systems*. ACM, 2017, pp. 48–57.

[44] W. Ali and H. Yun, “Protecting real-time gpu applications on integrated cpu-gpu soc platforms,” in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[45] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, “Rgem: A responsive gpgpu execution model for runtime engines,” in *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 2011, pp. 57–66.

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

[47] H. Zhou, G. Tong, and C. Liu, “Gpes: A preemptive execution system for gpgpu computing,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*. IEEE, 2015, pp. 87–97.

[48] H. Lee, J. Roh, and E. Seo, “A gpu kernel transactionization scheme for preemptive priority scheduling,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 202–213.

[49] C. Hartmann and U. Margull, “Gpuart-an application-based limited preemptive gpu real-time scheduler for embedded systems,” *Journal of Systems Architecture*, vol. 97, pp. 304–319, 2019.