Approximation Proofs of a Fast and Efficient List Scheduling Algorithm for Task-Based Runtime Systems on Multicores and GPUs

Olivier Beaumont, Lionel Eyraud-Dubois, Suraj Kumar

Approximation Proofs of a Fast and Efficient List Scheduling Algorithm for Task-Based Runtime Systems on Multicores and GPUs

Olivier Beaumont, Lionel Eyraud-Dubois and Suraj Kumar
Inria and University of Bordeaux
Bordeaux, France
Email: firstname.lastname@inria.fr

Abstract

In High Performance Computing, heterogeneity is now the norm with specialized accelerators like GPUs providing efficient computational power. The added complexity has led to the development of task-based runtime systems, which allow complex computations to be expressed as task graphs, and rely on scheduling algorithms to perform load balancing between all resources of the platforms. Developing good scheduling algorithms, even on a single node, and analyzing them can thus have a very high impact on the performance of current HPC systems. The special case of two types of resources (namely CPUs and GPUs) is of practical interest. HeteroPrio is such an algorithm which has been proposed in the context of fast multipole computations, and then extended to general task graphs with very interesting results. In this paper, we provide a theoretical insight on the performance of HeteroPrio, by proving approximation bounds compared to the optimal schedule in the case where all tasks are independent and for different platform sizes. Interestingly, this shows that spoliation allows to prove approximation ratios for a list scheduling algorithm on two unrelated resources, which is not possible otherwise. We also establish that almost all our bounds are tight. Additionally, we provide an experimental evaluation of HeteroPrio on real task graphs from dense linear algebra computation, which highlights the reasons explaining its good practical performance.

Keywords: List scheduling; Approximation proofs; Runtime systems; Heterogeneous scheduling; Dense linear algebra;

1 Introduction

Accelerators such as GPUs are more and more commonplace in processing nodes due to their massive computational power, usually beside multicores. When

trying to exploit both CPUs and GPUs, users face several issues. Indeed, several phenomena are added to the inherent complexity of the underlying NP-hard optimization problem.

First, multicores and GPUs are unrelated resources, in the sense that depending on the targeted kernel, the performance of the GPUs may be much higher, close or even worse than the performance of a CPU. In the literature, unrelated resources are known to make scheduling problems harder (see [1] for a survey on the complexity of scheduling problems, [2] for the specific simpler case of independent tasks scheduling and [3] for a recent survey in the case of CPU and GPU nodes). Second, the number of available architectures has increased dramatically with the combination of available resources (both in terms of multicores and accelerators). Therefore, it is almost impossible to develop optimized hand tuned kernels for all these architectures. Third, nodes have many shared resources (caches, buses) and exhibit complex memory access patterns (NUMA effects), that render the precise estimation of the duration of tasks and data transfers extremely difficult.

All these characteristics make it hard to design scheduling and resource allocation policies even on very regular kernels such as linear algebra. On the other hand, this situation favors dynamic strategies where decisions are made at runtime based on the state of the machine and on the knowledge of the application (to favor tasks that are close to the critical path for instance). In recent years, several task-based systems have been developed such as StarPU [4], StarSs [5], SuperMatrix [6], QUARK [7], XKaapi [8] or PaRSEC [9]. All these runtime systems model the application as a DAG, where nodes correspond to tasks and edges to dependencies between these tasks. At runtime, the scheduler knows (i) the state of the different resources (ii) the set of tasks that are currently processed by all non-idle resources (iii) the set of (independent) tasks whose all dependencies have been solved (iv) the location of all input data of all tasks (v) possibly an estimation of the duration of each task on each resource and of each communication between each pair of resources and (vi) possibly priorities associated to tasks that have been computed offline. Therefore, the scheduling problem consists in deciding, for an independent set of tasks, given the characteristics of these tasks on the different resources, where to place and to execute them. This paper is devoted to this specific problem.

On the theoretical side, several solutions have been proposed for this problem, including PTAS (see for instance [10]). Nevertheless, in the target application, dynamic schedulers must take their decisions at runtime and are themselves on the critical path of the application. This reduces the spectrum of possible algorithms to very fast ones, whose complexity to decide which task to execute next should be sublinear in the number of ready tasks.

Several scheduling algorithms have been proposed in this context and can be classified in several classes. The first class of algorithms is based on (variants of) HEFT [11], where the priority of tasks is computed based on their expected distance to the last node, with several possible metrics to define the expected durations of tasks (given that tasks can be processed on heterogeneous resources) and data transfers (given that input data may be located on different resources).

To the best of our knowledge there is not any approximation ratio for this class of algorithms on unrelated resources and Bleuse et al. [3] have exhibited an example on m CPUs and 1 GPU where HEFT algorithm achieves a makespan $\Omega(m)$ times worse the optimal. The second class of scheduling algorithms is based on more sophisticated ideas that aim at minimizing the makespan of the set of ready tasks (see for instance [3]). In this class of algorithms, the main difference lies in the compromise between the quality of the scheduling algorithm (expressed as its approximation ratio when scheduling independent tasks) and its cost (expressed as the complexity of the scheduling algorithm). At last, a third class of algorithms has recently been proposed (see for instance [12]), in which scheduling decisions are based on the affinity between tasks and resources, *i.e.* try to process the tasks on the best suited resource for it.

In this paper, we concentrate on HeteroPrio that belongs to the third class and that is described in details in Section 2. More specifically, we prove that HeteroPrio combines the best of all worlds. Indeed, after discussing the related work in Section 3 and introducing notations and general results in Section 4, we first prove that contrarily to HEFT variants, HeteroPrio achieves a bounded approximation ratio in Section 5 and we provide a set of proved and tight approximation results, depending on the number of CPUs and GPUs in the node. At last, we provide in Section 6 a set of experimental results showing that, besides its very low complexity, HeteroPrio achieves a better performance than the other schedulers based either on HEFT or on an approximation algorithm for independent tasks scheduling. Concluding remarks are given in Section 7.

2 HeteroPrio Principle

2.1 Affinity Based Scheduling

HeteroPrio has been proposed in the context of task-based runtime systems responsible for allocating tasks onto heterogeneous nodes typically consisting of a few CPUs and GPUs [13].

Historically, in most systems, tasks are ordered by priorities (computed offline) and the highest priority ready task is allocated on the resource that is expected to complete it first, given the estimation of the transfer times of its input data and the expected processing time of this task on this resource. These systems have shown some limits in strongly heterogeneous and unrelated systems, what is typically the case of nodes consisting of both CPUs and GPUs. Indeed, the relative efficiency of accelerators, that we call the affinity in what follows, strongly differs from one task to another. Let us for instance consider the case of Cholesky factorization, where 4 types of tasks (kernels DPOTRF, DTRSM, DSYRK and DGEMM) are involved. The acceleration factors are depicted in Table 1.

In all what follows, acceleration factor is always defined as the ratio between the processing time on a CPU and on a GPU, so that the acceleration factor may be smaller than 1. From this table, we can extract the main features that will influence our model. The acceleration factor strongly depends on the kernel. Some kernels, like DSYRK and DGEMM are almost 30 times faster on GPUs,

	DPOTRF	DTRSM	DSYRK	DGEMM
CPU time / GPU time	1.72	8.72	26.96	28.80

Tab. 1: Acceleration factors for Cholesky kernels (size 960)

DPOTRF are only slightly accelerated. Based on this observation, a different class of runtime schedulers for task based systems has been developed, in which the affinity between tasks and resources plays the central role. HeteroPrio belongs to this class. In these systems, when a resource becomes idle, it selects among the ready tasks the one for which it has a maximal affinity. For instance, in the case of Cholesky factorization, among the ready tasks, CPUs will prefer DPOTRF to DTRSM to DSYRK to DGEMM and GPUs will prefer DGEMM to DSYRK to DTRSM to DPOTRF. HeteroPrio allocation strategy has been studied in the context of StarPU for several linear algebra kernels and it has been proved experimentally that it enables to achieve a better utilization of slow resources than other strategies based on the minimization of the completion time. Nevertheless, in order to be efficient, HeteroPrio must be associated to a spoliation mechanism. Indeed, in above description, nothing prevents the slow resource to execute a task for which it can be arbitrarily badly suited, thus leading to arbitrarily bad results. Therefore, when a fast resource is idle and would be able to restart a task already started on a slow resource and to finish it earlier than on the slow resource, then the task is spoliated and restarted on the fast resource. Note that this mechanism does not correspond to preemption since all the progress made on the slow resource is lost. It is therefore less efficient than preemption but it can be implemented in practice, what is not the case of preemption on heterogeneous resources like CPUs and GPUs.

In what follows, since task based runtime systems see a set of independent tasks, we will concentrate on this problem and we will prove approximation ratios for HeteroPrio under several scenarios for the composition of the heterogeneous node (namely 1 GPU and 1 CPU, 1 GPU and several CPUs and several GPUs and several CPUs).

2.2 HeteroPrio Algorithm for a set of Independent Tasks

When priorities are associated with tasks then Line 1 of Algorithm 1 takes them into account for breaking ties among tasks with the same acceleration factor and put highest (resp. lowest) priority task first in the scheduling queue for acceleration factor≥ 1 (resp. < 1). Queue of ready tasks in Algorithm 1 can be implemented as a heap. Therefore, time complexity of Algorithm 1 would be $\mathcal{O}(Nlog(N))$, where N is the number of ready tasks.

3 Related Works

The problem considered in this paper is a special case of the standard unrelated scheduling problem $R||C_{\max}$. Lenstra et al [2] proposed a PTAS for the general

Algorithm 1: The HeteroPrio Algorithm for a set of independent tasks

```
Sort Ready tasks in queue Q by non-increasing acceleration factors
while all tasks did not complete do
    if all workers are busy then
        continue
    end if
    Select an idle worker W
    if Q ≠ ∅ then
        Remove a task T from beginning of Q if W is a GPU worker
        otherwise from end of Q
        W starts processing T
    else
        Consider tasks running on the other type of resource in decreasing
        order of their expected completion time. If the expected completion
        time of T running on a worker W′ can be improved on W, T is
        spoliated and W starts processing T.
    end if
end while
```

problem with a fixed number of machines, and a 2-approximation algorithm, based on the rounding of the optimal solution of the linear program which describes the preemptive version of the problem. This result has recently been improved [14] to a $2-\frac{1}{m}$ approximation. However, the time complexity of these general algorithms is too high to allow using them in the context of runtime systems.

The more specialized case with a small number of types of resources has been studied in [10] and a PTAS has been proposed, which also contains a rounding phase whose complexity makes it impractical, even for 2 different types of resources. Greedy approximation algorithms for the online case have been proposed by Imreh on two different types of resources [15]. These algorithms have linear complexity, however most of their decisions are based on comparing task execution times on both types of resources and not on trying to balance the load. The result is that in the practical test cases of interest to us, almost all tasks are scheduled on the GPUs and the performance is significantly worse. Finally, Bleuse et al [3, 16] have proposed algorithms with varying approximation factors ($\frac{4}{3}$, $\frac{3}{2}$ and 2) based on dynamic programming and dual approximation techniques. These algorithms have better approximation ratios than the ones proved in this paper, but their time complexity is higher. Furthermore, as we show in Section 6, their actual performance is not as good when used iteratively on the set of ready tasks in the context of task graph scheduling. We also exhibit that HeteroPrio performs better on average than above mentioned algorithms, despite its higher worst case approximation ratio.

In homogeneous scheduling, list algorithms (*i.e.* algorithms that never leave a resource idle if there exists a ready task) are known to have good practical

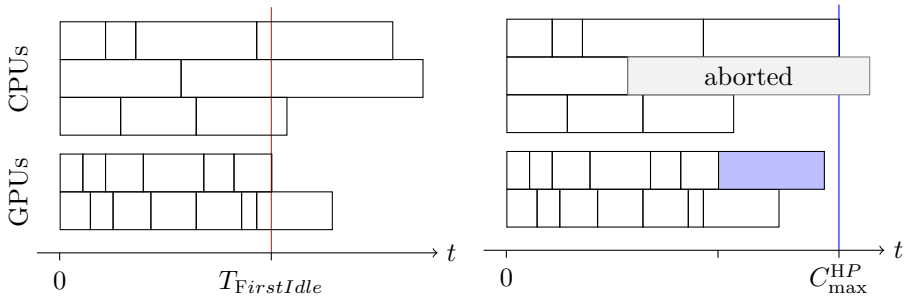
performance. In the context of heterogeneous scheduling, it is well known that list scheduling algorithms cannot achieve an approximation guarantee. Indeed, even with two resources and two tasks, if one resource is much slower than the other, it can be arbitrarily better to leave it idle and to execute both tasks on the fast resource. The HeteroPrio algorithm considered in this paper is based on a list algorithm, but the use of spoliation (see Section 2.2) avoids this problem.

4 Notations and First Results

4.1 General Notations

In this paper, we study the theoretical guarantee of HeteroPrio for a set of independent tasks. In the scheduling problem that we consider, the input is thus a platform of n GPUs and m CPUs and a set $\mathcal{I}$ of independent tasks, where task T_i has processing time p_i on CPU and q_i on GPU, and the goal is to schedule those tasks on the resources so as to minimize the makespan. We define the acceleration factor of task T_i as $\rho_i = \frac{p_i}{q_i}$ and $C_{\max}^{Opt}(\mathcal{I})$ denotes the optimal makespan of set $\mathcal{I}$.

To analyze the behavior of HeteroPrio, it is useful to consider the list schedule obtained before any spoliation attempt. We will denote this schedule $\mathcal{S}_{\text{HP}}^{\text{NS}}$, and the final HeteroPrio schedule is denoted $\mathcal{S}_{\text{HP}}$. Figure 1 shows $\mathcal{S}_{\text{HP}}^{\text{NS}}$ and $\mathcal{S}_{\text{HP}}$ for a set of independent tasks $\mathcal{I}$. We define $T_{\text{F}irstIdle}$ as the first time any worker is idle in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, this is also the first time any spoliation can occur. Therefore after time $T_{\text{F}irstIdle}$, each worker executes at most one task in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. Finally, we define $C_{\max}^{HP}(\mathcal{I})$ as the makespan of $\mathcal{S}_{\text{HP}}$ on instance $\mathcal{I}$.



(a) Schedule $\mathcal{S}_{\text{HP}}^{\text{NS}}$ (b) $\mathcal{S}_{\text{HP}}$, with a spoliated task

Fig. 1: Example of a HeteroPrio schedule

4.2 Area Bound

In this section, we present and characterize a lower bound on the optimal makespan. This lower bound is obtained by assuming that tasks are divisible, *i.e.* can be processed in parallel on any number of resources. More specifically,

any fraction x_i of task T_i is allowed to be processed on CPUs, and this fraction overall consumes CPU resources for $x_i p_i$ time units. Then, the lower bound $AreaBound(\mathcal{I})$ for a set of tasks $\mathcal{I}$ on m CPUs and n GPUs is the solution (in rational numbers) of the following linear program.

Minimize $AreaBound(\mathcal{I})$ such that

$$\sum_{i \in I} x_i p_i \leq m \cdot AreaBound(\mathcal{I}) \tag{1}$$

$$\sum_{i \in I} (1 - x_i) q_i \leq n \cdot AreaBound(\mathcal{I}) \tag{2}$$

$$0 \leq x_i \leq 1$$

Since any valid solution to the scheduling problem can be converted into a solution of this linear program, it is clear that $AreaBound(\mathcal{I}) \leq C_{\max}^{Opt}(\mathcal{I})$. Another immediate bound on the optimal is $\forall T \in \mathcal{I}, \min(p_T, q_T) \leq C_{\max}^{Opt}(\mathcal{I})$. By contradiction and with simple exchange arguments, one can prove the following two lemmas.

Lemma 1. *In the area bound solution, the completion time on each class of resources is the same, i.e. constraints* (1) *and* (2) *are both equalities.*

Proof. Let us assume that one of the inequality constraints of area solution is not tight. Without loss of generality, let us assume that Constraint (1) is not tight. Then some load from the GPUs can be transferred to the CPUs which in turn decreases the value of $AreaBound(\mathcal{I})$. This achieves the proof of the Lemma. □

Lemma 2. *In $AreaBound(\mathcal{I})$, the assignment of tasks is based on the acceleration factor, i.e. $\exists k > 0$ such that $\forall i, x_i < 1 \Rightarrow \rho_i \geq k$ and $x_i > 0 \Rightarrow \rho_i \leq k$.*

Proof. Let us assume $\exists (T_1, T_2)$ such that (i) T_1 is partially processed on GPUs (*i.e.*, $x_1 < 1$), (ii) T_2 is partially processed on CPUs (*i.e.*, $x_2 > 0$) and (iii) $\rho_1 < \rho_2$.

Let WC and WG denote respectively the overall work on CPUs and GPUs in $AreaBound(\mathcal{I})$. If we transfer a fraction $0 < \epsilon_2 < min(x_2, \frac{(1-x_1)p_1}{p_2})$ of T_2 work from CPU to GPU and a fraction $\frac{\epsilon_2 q_2}{q_1} < \epsilon_1 < \frac{\epsilon_2 p_2}{p_1}$ of T_1 work from GPU to CPU, the overall loads WC' and WG' become

$$WC' = WC + \epsilon_1 p_1 - \epsilon_2 p_2$$
$$WG' = WG - \epsilon_1 q_1 + \epsilon_2 q_2$$

Since $\frac{p_1}{p_2} < \frac{\epsilon_2}{\epsilon_1} < \frac{q_1}{q_2}$, then both $WC' < WC$ and $WG' < WG$ hold true, and hence the $AreaBound(\mathcal{I})$ is not optimal. Therefore, $\exists$ a positive constant k such that $\forall i$ on GPU, $\rho_i \geq k$ and $\forall i$ on CPU, $\rho_i \leq k$. □

4.3 Summary of Approximation Results

This paper presents several approximation results depending on the number of CPUs and GPUs. The following table presents a quick overview of the main results proven in Section 5.

(#CPUs, #GPUs)	Approximation ratio	Worst case ex.
(1,1)	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$
(m,1)	$\frac{3+\sqrt{5}}{2}$	$\frac{3+\sqrt{5}}{2}$
(m,n)	$2+\sqrt{2} \approx 3.41$	$2+\frac{2}{\sqrt{3}} \approx 3.15$

5 Proof of HeteroPrio Approximation Results

5.1 General Lemmas

The following lemma gives a characterization of the work performed by HeteroPrio at the beginning of the execution, and shows that HeteroPrio performs as much work as possible when all resources are busy. At any instant t, let us define $\mathcal{I}'(t)$ as the sub-instance of $\mathcal{I}$ composed of the fractions of tasks that have not been entirely processed at time t by HeteroPrio. Then, a schedule beginning like HeteroPrio (until time t) and ending like $AreaBound(\mathcal{I}'(t))$ completes in $AreaBound(\mathcal{I})$.

Lemma 3. *At any time $t \leq T_{FirstIdle}$ in $\mathcal{S}^{NS}_{HP}$,*

$$t + AreaBound(\mathcal{I}'(t)) = AreaBound(\mathcal{I})$$

Proof. HeteroPrio assigns tasks based on their acceleration factors. Therefore, at instant t, $\exists k_1 \leq k_2$ such that (i) all tasks (at least partially) processed on GPUs have an acceleration factor larger than k_2, (ii) all tasks (at least partially) allocated on CPUs have an acceleration factor smaller than k_1 and (iii) all tasks not assigned yet have an acceleration factor between k_1 and k_2.

After t, $AreaBound(\mathcal{I}')$ satisfies Lemma 2, and thus $\exists k$ with $k_1 \leq k \leq k_2$ such that all tasks of $\mathcal{I}'$ with acceleration factor larger than k are allocated on GPUs and all tasks of $\mathcal{I}'$ with acceleration factor smaller than k are allocated on CPUs.

Therefore, combining above results before and after t, the assignment $\mathcal{S}$ beginning like HeteroPrio (until time t) and ending like $AreaBound(\mathcal{I}'(t))$ satisfies the following property: $\exists k > 0$ such that all tasks of $\mathcal{I}$ with acceleration factor larger than k are allocated on GPUs and all tasks of $\mathcal{I}$ with acceleration factor smaller than k are allocated on CPUs. This assignment $\mathcal{S}$, whose completion time on both CPUs and GPUs (thanks to Lemma 1) is $t + AreaBound(\mathcal{I}')$ clearly defines a solution of the fractional linear program defining the area bound solution, so that $t + AreaBound(\mathcal{I}') \geq AreaBound(\mathcal{I})$.

Similarly, $AreaBound(\mathcal{I})$ satisfies both Lemma 2 with some value k' and Lemma 1 so that in $AreaBound(\mathcal{I})$, both CPUs and GPUs complete their work simultaneously. If $k' < k$, more work is assigned to GPUs in $AreaBound(\mathcal{I})$

than in $\mathcal{S}$, so that, by considering the completion time on GPUs, we get $AreaBound(\mathcal{I}) \geq t + AreaBound(\mathcal{I}')$. Similarly, if $k' > k$, by considering the completion time on CPUs, we get $AreaBound(\mathcal{I}) \geq t + AreaBound(\mathcal{I}')$. This achieves the proof of Lemma 3.

□

Since $AreaBound(\mathcal{I})$ is a lower bound on $C^{Opt}_{\max}(\mathcal{I})$, the above lemma implies that
(i) at any time $t \leq T_{FirstIdle}$ in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$, $t + AreaBound(\mathcal{I}'(t)) \leq C^{Opt}_{\max}(\mathcal{I})$,
(ii) $T_{FirstIdle} \leq C^{Opt}_{\max}(\mathcal{I})$, and thus all tasks start before $C^{Opt}_{\max}(\mathcal{I})$ in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$,
(iii) if $\forall i \in \mathcal{I}, \max(p_i, q_i) \leq C^{Opt}_{\max}(\mathcal{I})$, then $C^{HP}_{\max}(\mathcal{I}) \leq 2C^{Opt}_{\max}(\mathcal{I})$.

Another interesting characteristic of HeteroPrio is that spoliation can only take place from one type of resource to the other. Indeed, since assignment in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$ is based on the acceleration factors of the tasks, and since a task can only be spoliated if it can be processed faster on the other resource, we get the following lemmas.

Lemma 4. *If, in $\mathcal{S}^{NS}_{HP}$, a resource processes a task whose execution time is not larger on the other resource, then no task is spoliated from the other resource.*

Proof. Without loss of generality let us assume that there exists a task T executed on a CPU in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$, such that $p_T \geq q_T$. We prove that in that case, there is no spoliated task on CPUs, which is the same thing as there being no aborted task on GPUs.

T is executed on a CPU in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$, and $\frac{p_T}{q_T} \geq 1$, therefore from HeteroPrio principle, all tasks on GPUs in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$ have an acceleration factor at least $\frac{p_{T'}}{q_{T'}} \geq 1$. Non spoliated tasks running on GPUs after $T_{FirstIdle}$ are candidates to be spoliated by the CPUs. But for each of these tasks, the execution time on CPU is at least as large as the execution time on GPU. It is thus not possible for an idle CPU to spoliate any task running on GPUs, because this task would not complete earlier on the CPU. □

Lemma 5. *In HeteroPrio, if a resource executes a spoliated task then no task is spoliated from this resource.*

Proof. Without loss of generality let us assume that T is a spoliated task executed on a CPU. From the HeteroPrio definition, $p_T < q_T$. It also indicates that T was executed on a GPU in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$ with $q_T \geq p_T$. By Lemma 4, CPUs do not have any aborted task due to spoliation. □

Finally, we will also rely on the following lemma, that gives the worst case performance of a list schedule when all tasks lengths are large (*i.e.* $> C^{Opt}_{\max}$) on one type of resource.

Lemma 6. *Let $\mathcal{B} \subseteq \mathcal{I}$ such that the execution time of each task of $\mathcal{B}$ on one resource is larger than $C^{Opt}_{\max}(\mathcal{I})$, then any list schedule of $\mathcal{B}$ on $k \geq 1$ resources of the other type has length at most $(2 - \frac{1}{k})C^{Opt}_{\max}(\mathcal{I})$.*

Proof. Without loss of generality, let us assume that the processing time of each task of set $\mathcal{B}$ on CPU is larger than $C_{\max}^{Opt}(\mathcal{I})$. All these tasks must therefore be processed on the GPUs in an optimal solution. If scheduling this set $\mathcal{B}$ on k GPUs can be done in time C, then $C \leq C_{\max}^{Opt}(\mathcal{I})$. The standard list scheduling result from Graham implies that the length of any list schedule of the tasks of $\mathcal{B}$ on GPUs is at most $(2-\frac{1}{k})C \leq (2-\frac{1}{k})C_{\max}^{Opt}(\mathcal{I})$. □

5.2 Approximation Ratio with 1 GPU and 1 CPU

Thanks to the above lemmas, we are able to prove an approximation ratio of $\phi = \frac{1+\sqrt{5}}{2}$ for HeteroPrio when the node is composed of 1 CPU and 1 GPU. We will also prove that this result is the best achievable by providing a task set $\mathcal{I}$ for which the approximation ratio of HeteroPrio is ϕ.

Theorem 7. *For any instance $\mathcal{I}$ with 1 CPU and 1 GPU, $C_{\max}^{HP}(\mathcal{I}) \leq \phi C_{\max}^{Opt}(\mathcal{I})$.*

Proof. Without loss of generality, let us assume that the first idle time (at instant $T_{FirstIdle}$) occurs on the GPU and the CPU is processing the last remaining task T. We will consider two main cases, depending on the relative values of $T_{FirstIdle}$ and $(\phi-1)C_{\max}^{Opt}$.

– $T_{FirstIdle} \leq (\phi-1)C_{\max}^{Opt}$.

In $\mathcal{S}_{\text{HP}}^{\text{NS}}$, the finish time of task T is at most $T_{FirstIdle} + p_T$. If task T is spoliated by the GPU, its execution time is $T_{FirstIdle} + q_T$. In both cases, the finish time of task T is at most $T_{FirstIdle} + \min(p_T, q_T) \leq (\phi-1)C_{\max}^{Opt} + C_{\max}^{Opt} = \phi C_{\max}^{Opt}$.

– $T_{FirstIdle} > (\phi-1)C_{\max}^{Opt}$.

If T ends before $\phi C_{\max}^{Opt}$ on the CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, since spoliation can only improve the completion time, this ends the proof of the theorem. In what follows, we assume that the completion time of T on the CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ is larger than $\phi C_{\max}^{Opt}(\mathcal{I})$, as depicted in Figure 2.

It is clear that T is the only unfinished task after $C_{\max}^{Opt}$. Let us denote by α the fraction of T processed after $C_{\max}^{Opt}$ on the CPU. Then $\alpha p_T > (\phi-1)C_{\max}^{Opt}$ since T ends after $\phi C_{\max}^{Opt}$ by assumption. Lemma 3 applied at instant $t = T_{FirstIdle}$ implies that the GPU is able to process the fraction α of T by $C_{\max}^{Opt}$ (see Figure 3) while starting this fraction at $T_{FirstIdle} \geq (\phi-1)C_{\max}^{Opt}$ so that $\alpha q_T \leq (1-(\phi-1))C_{\max}^{Opt} = (2-\phi)C_{\max}^{Opt}$. Therefore, the acceleration factor of T is at least $\frac{\phi-1}{2-\phi} = \phi$. Since HeteroPrio assigns tasks on the GPU based on their acceleration factors, all tasks in $\mathcal{S}$ assigned to the GPU also have an acceleration factor at least ϕ.

Let us now prove that the GPU is able to process $\mathcal{S} \bigcup \{T\}$ in time $\phi C_{\max}^{Opt}$. Let us split $\mathcal{S} \bigcup \{T\}$ into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ depending on whether the tasks of $\mathcal{S} \bigcup \{T\}$ are processed on the GPU ($\mathcal{S}_1$) or on the CPU ($\mathcal{S}_2$) in the optimal solution. By construction, the processing time of $\mathcal{S}_1$ on the GPU is at most $C_{\max}^{Opt}$ and the processing of $\mathcal{S}_2$ on the CPU takes at most $C_{\max}^{Opt}$. Since the acceleration factor of tasks of $\mathcal{S}_2$ is larger than ϕ, the processing time of tasks of $\mathcal{S}_2$ on the GPU is at most $C_{\max}^{Opt}/\phi$ and the overall execution of $\mathcal{S} \bigcup \{T\}$ takes at most $C_{\max}^{Opt} + C_{\max}^{Opt}/\phi = \phi C_{\max}^{Opt}$, what ends the proof of the theorem. □

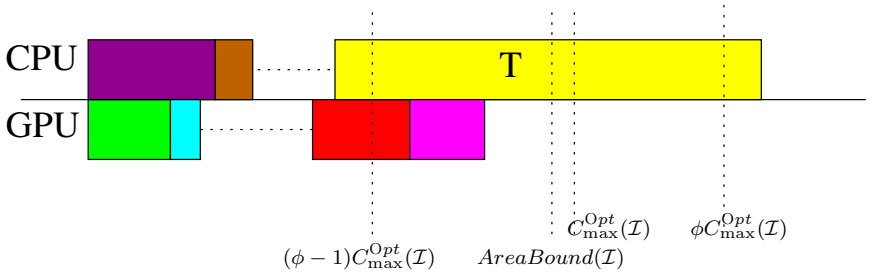


Fig. 2: Situation where T ends on CPU after $\phi C_{\max}^{Opt}(\mathcal{I})$.

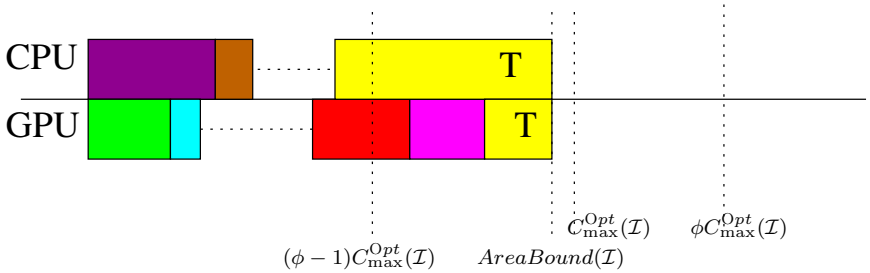


Fig. 3: Area bound consideration to bound the acceleration factor of T.

Theorem 8. *The bound of Theorem 7 is tight, i.e. there exists an instance $\mathcal{I}$ with 1 CPU and 1 GPU for which HeteroPrio achieves a ratio of ϕ with respect to the optimal solution.*

Proof. Let us consider the instance $\mathcal{I}$ consisting of 2 tasks X and Y, with $p_X = \phi$, $q_X = 1$, $p_Y = 1$ and $q_Y = \frac{1}{\phi}$, such that $\rho_X = \rho_Y = \phi$.

The minimum length of task X is 1, so that $C_{\max}^{Opt} \geq 1$. Moreover, allocating X on the GPU and Y on the CPU leads to a makespan of 1, so that $C_{\max}^{Opt} \leq 1$ and finally $C_{\max}^{Opt} = 1$.

On the other hand, consider the following valid HeteroPrio schedule, where CPU first selects X and the GPU first selects Y. GPU becomes available at instant $\frac{1}{\phi} = \phi - 1$ but does not spoliate task X since it cannot complete X earlier than its expected completion time on the CPU. Therefore, the completion time of HeteroPrio is $\phi = \phi C_{\max}^{Opt}$. □

5.3 Approximation Ratio with 1 GPU and m CPUs

In the case of a single GPU and m CPUs, the approximation ratio of HeteroPrio becomes $1 + \phi = \frac{3+\sqrt{5}}{2}$, as proved in Theorem 9 and this bound is tight (asymptotically when m becomes large) as proved in Theorem 11.

Theorem 9. *HeteroPrio achieves an approximation ratio of $(1 + \phi) = \frac{3+\sqrt{5}}{2}$ for any instance $\mathcal{I}$ on m CPUs and 1 GPU.*

Proof. Let us assume by contradiction that there exists a task T whose completion time is larger than $(1+\phi)C_{\max}^{Opt}$. We know that all tasks start before $C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. If T is executed on the GPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, then $q_T > C_{\max}^{Opt}$ and thus $p_T \leq C_{\max}^{Opt}$. Since at least one CPU is idle at time $T_{FirstIdle}$, T should have been spoliated and processed by $2C_{\max}^{Opt}$.

We know that T is processed on a CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, and finishes later than $(1+\phi)C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}$. Let us denote by $\mathcal{S}$ the set of all tasks spoliated by the GPU (from a CPU to the GPU) before considering T for spoliation in the execution of HeteroPrio and let us denote by $\mathcal{S}' = \mathcal{S} \bigcup \{T\}$. The following lemma will be used to complete the proof.

Lemma 10. *The following holds true*
(i) $p_i > C_{\max}^{Opt}$ for all tasks i of $\mathcal{S}'$,
(ii) the acceleration factor of T is at least ϕ,
(iii) the acceleration factor of tasks running on the GPU in $\mathcal{S}_{HP}^{NS}$ is at least ϕ.

Proof. of Lemma 10. Since all tasks start before $T_{FirstIdle} \leq C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, and since T finishes after $(1+\phi)C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, then $p_T > \phi C_{\max}^{Opt}$. Since HeteroPrio performs spoliation of tasks in decreasing order of their completion time, the same applies to all tasks of $\mathcal{S}'$: $\forall i \in \mathcal{S}', p_i > \phi C_{\max}^{Opt}$, and thus $q_i \leq C_{\max}^{Opt}$.
Since $p_T > \phi C_{\max}^{Opt}$ and $q_T \leq C_{\max}^{Opt}$, then $\rho_T > \phi$. Since T is executed on a CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, all tasks executed on GPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ have an acceleration factor at least ϕ. □

Since T is processed on the CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ and $p_T > q_T$, Lemma 4 applies and no task is spoliated from the GPU. Let A be the set of tasks running on GPU right after $T_{FirstIdle}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. We consider only one GPU, therefore $|A| \leq 1$.

1. If $A = \{a\}$ with $q_a \leq (\phi-1)C_{\max}^{Opt}$, then Lemma 6 applies to $\mathcal{S}'$ (with $n=1$) and the overall completion time is $\leq T_{FirstIdle} + q_A + C_{\max}^{Opt} \leq (\phi+1)C_{\max}^{Opt}$.
2. If $A = \{a\}$ with $q_a > (\phi-1)C_{\max}^{Opt}$, since $\rho_a > \phi$ by Lemma 10, $p_a > \phi(\phi-1)C_{\max}^{Opt} = C_{\max}^{Opt}$. Lemma 6 applies to $\mathcal{S}' \bigcup A$, so that the overall completion time is bounded by $T_{FirstIdle} + C_{\max}^{Opt} \leq 2C_{\max}^{Opt}$.
3. If $A = \emptyset$, Lemma 6 applies to $\mathcal{S}'$ and get $C_{\max}^{HP}(\mathcal{I}) \leq T_{FirstIdle} + C_{\max}^{Opt} \leq 2C_{\max}^{Opt}$.

Therefore, in all cases, the completion time of task T is at most $(\phi+1)C_{\max}^{Opt}$, what ends the proof of Theorem 9. □

Theorem 11. *Theorem 9 is tight, i.e. for any $\delta > 0$, there exists an instance $\mathcal{I}$ such that $C_{\max}^{HP}(\mathcal{I}) \geq (\phi + 1 - \delta)C_{\max}^{Opt}(\mathcal{I})$.*

Proof. $\forall \epsilon > 0$, let $\mathcal{I}$ denote the following set

Task	CPU Time	GPU Time	# of tasks	accel ratio
T_1	1	$1/\phi$	1	ϕ
T_2	ϕ	1	1	ϕ
T_3	ϵ	ϵ	$(mx)/\epsilon$	1
T_4	$\epsilon\phi$	ϵ	x/ϵ	ϕ

where $x = (m-1)/(m+\phi)$.

The minimum length of task T_2 is 1, so that $C_{\max}^{Opt} \geq 1$. Moreover, if T_1, T_3 and T_4 are scheduled on CPUs and T_2 on the GPU (this is possible if ϵ is small enough), then the completion time is 1, so that $C_{\max}^{Opt} = 1$.
Consider the following valid HeteroPrio schedule. The GPU first selects tasks from T_4 and the CPUs first select tasks from T_3. All resources become available at time x. Now, the GPU selects task T_1 and one of the CPUs selects task T_2, with a completion time of $x + \phi$. The GPU becomes available at $x + 1/\phi$ but does not spoliate T_2 since it would not finish before $x + 1/\phi + 1 = x + \phi$. The makespan of HeteroPrio is thus $x + \phi$, and since x tends towards 1 when m becomes large, the approximation ratio of HeteroPrio on this instance tends towards $1 + \phi$. □

5.4 Approximation Ratio with n GPUs and m CPUs

In the most general case of n GPUs and m CPUs, the approximation ratio of HeteroPrio is at most $2+\sqrt{2}$, as proved in Theorem 12. To establish this result, we rely on the same techniques as in the case of a single GPU, but the result of Lemma 6 is weaker for $n > 1$, what explains that the approximation ratio is larger than for Theorem 9. We have not been able to prove, as previously, that this bound is tight, but we provide in Theorem 14 a family of instances for which the approximation ratio is arbitrarily close to $2 + \frac{2}{\sqrt{3}}$.

Theorem 12. $\forall \mathcal{I}$, $C_{\max}^{HP}(\mathcal{I}) \leq (2+\sqrt{2}) C_{\max}^{Opt}(\mathcal{I})$.

Proof. We prove this by contradiction. Let us assume that there exists a task T whose completion time in $\mathcal{S}_{\text{HP}}$ is larger than $(2+\sqrt{2})C_{\max}^{Opt}$. Without loss of generality, we assume that T is executed on a CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. In the rest of the proof, we denote by $\mathcal{S}$ the set of all tasks spoliated by GPUs in the HeteroPrio solution, and $\mathcal{S}' = \mathcal{S} \cup \{T\}$. The following lemma will be used to complete the proof.

Lemma 13. *The following holds true*
(i) $\forall i \in \mathcal{S}', p_i > C_{\max}^{Opt}$
(ii) $\forall T'$ *processed on GPU in* $\mathcal{S}_{HP}^{NS}$, $\rho_{T'} \geq 1 + \sqrt{2}$.

Proof. of Lemma 13. In $\mathcal{S}_{\text{HP}}^{\text{NS}}$, all tasks start before $T_{FirstIdle} \leq C_{\max}^{Opt}$. Since T ends after $(2+\sqrt{2})C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ (since spoliation can only improve the completion time), then $p_T > (1+\sqrt{2})C_{\max}^{Opt}$. The same applies to all spoliated tasks that complete after T in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. If T is not considered for spoliation, no task that complete before T in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ is spoliated, and the first result holds. Otherwise, let s_T denote the instant at which T is considered for spoliation. The completion time of T in $\mathcal{S}_{\text{HP}}$ is at most $s_T + q_T$, and since $q_T \leq C_{\max}^{Opt}$, $s_T \geq (1+\sqrt{2})C_{\max}^{Opt}$. Since HeteroPrio handles tasks for spoliation in decreasing order of their completion time in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, tasks T' is spoliated after T has been considered and not finished at time s_T, and thus $p_{T'} > \sqrt{2}C_{\max}^{Opt}$.
Since $p_T > (1+\sqrt{2})C_{\max}^{Opt}$ and $q_T \leq C_{\max}^{Opt}$, then $\rho_T \geq (1+\sqrt{2})$. Since T is

executed on a CPU in $\mathcal{S}^{\text{NS}}_{\text{HP}}$, all tasks executed on GPU in $\mathcal{S}^{\text{NS}}_{\text{HP}}$ have acceleration factor at least $1+\sqrt{2}$. □

Let $\mathcal{A}$ be the set of tasks executed on GPUs after time $T_{\text{FirstIdle}}$ in $\mathcal{S}^{\text{NS}}_{\text{HP}}$. We partition $\mathcal{A}$ into two sets $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\forall i \in \mathcal{A}_1, q_i \leq \frac{C^{\text{Opt}}_{\max}}{\sqrt{2}+1}$ and $\forall i \in \mathcal{A}_2, q_i > \frac{C^{\text{Opt}}_{\max}}{\sqrt{2}+1}$.

Since there are n GPUs, $|\mathcal{A}_1| \leq |\mathcal{A}| \leq n$. We consider the schedule induced by HeteroPrio on the GPUs with the tasks $\mathcal{A} \bigcup \mathcal{S}'$ (if T is spoliated, this schedule is actually returned by HeteroPrio, otherwise this is what HeteroPrio builds when attempting to spoliate task T). This schedule is not worse than a schedule that processes all tasks from $\mathcal{A}_1$ starting at time $T_{\text{FirstIdle}}$, and then performs any list schedule of all tasks from $\mathcal{A}_2 \bigcup \mathcal{S}'$. Since $|\mathcal{A}_1| \leq n$, the first part takes time at most $\frac{C^{\text{Opt}}_{\max}}{\sqrt{2}+1}$. For all T_i in $\mathcal{A}_2$, $\rho_i \geq 1+\sqrt{2}$ and $q_i > \frac{C^{\text{Opt}}_{\max}(\mathcal{I})}{\sqrt{2}+1}$ imply $p_i > C^{\text{Opt}}_{\max}$. We can thus apply Lemma 6 to $\mathcal{A}_2 \bigcup \mathcal{S}'$ and the second part takes time at most $2C^{\text{Opt}}_{\max}$. Overall, the completion time on GPUs is bounded by $T_{\text{FirstIdle}} + \frac{C^{\text{Opt}}_{\max}}{\sqrt{2}+1} + (2-\frac{1}{n})C^{\text{Opt}}_{\max} < C^{\text{Opt}}_{\max} + (\sqrt{2}-1)C^{\text{Opt}}_{\max} + 2C^{\text{Opt}}_{\max} = (\sqrt{2}+2)C^{\text{Opt}}_{\max}$, which is a contradiction. □

Theorem 14. *The approximation ratio of HeteroPrio is at least* $2+\frac{2}{\sqrt{3}} \simeq 3.15$.

Proof. We consider an instance $\mathcal{I}$, with $n = 6k$ GPUs and $m = n^2$ CPUs, containing the following tasks.

Task	CPU Time	GPU Time	# of tasks	accel ratio
T_1	n	$\frac{n}{r}$	n	r
T_2	$\frac{rn}{3}$	see below	see below	$\frac{r}{3} \leq \rho \leq r$
T_3	1	1	mx	1
T_4	r	1	nx	r

where $x = \frac{(m-n)}{m+nr}n$ and r is the solution of the equation $\frac{n}{r} + 2n - 1 = \frac{nr}{3}$. Note that the highest acceleration factor is r and the lowest is 1 since $r > 3$. The set T_2 contains tasks with the following execution time on GPU,
(i) one task of length $n = 6k$, (ii) for all $0 \leq i \leq 2k-1$, six tasks of length $2k+i$.

This set T_2 of tasks can be scheduled on n GPUs in time n (see Figure 4). $\forall 1 \leq i < k$, each of the six tasks of length $2k+i$ can be combined with one of the six tasks of length $2k+(2k-i)$, occupying $6(k-1)$ processors; the tasks of length $3k$ can be combined together on 3 processors, and there remains 3 processors for the six tasks of length $2k$ and the task of length $6k$. On the other hand, the worst list schedule may achieve makespan $2n-1$ on the n GPUs. $\forall 0 \leq i \leq k-1$, each of the six tasks of length $2k+i$ is combined with one of the six tasks of length $4k-i-1$, which occupies all $6k$ processors until time $6k-1$, then the task of length $6k$ is executed. The fact that there exists a set of tasks for which the makespan of the worst case list schedule is almost twice

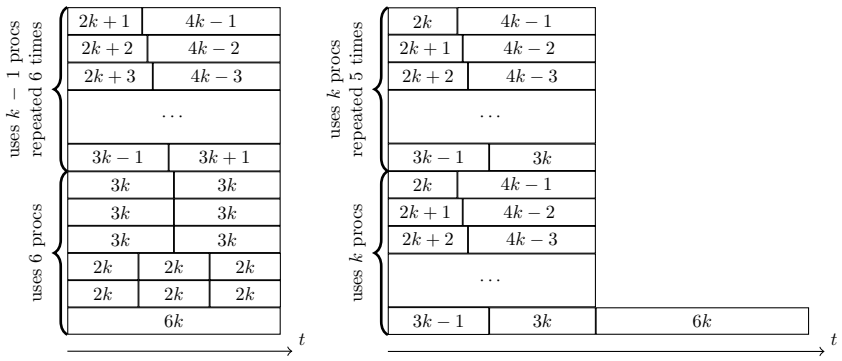
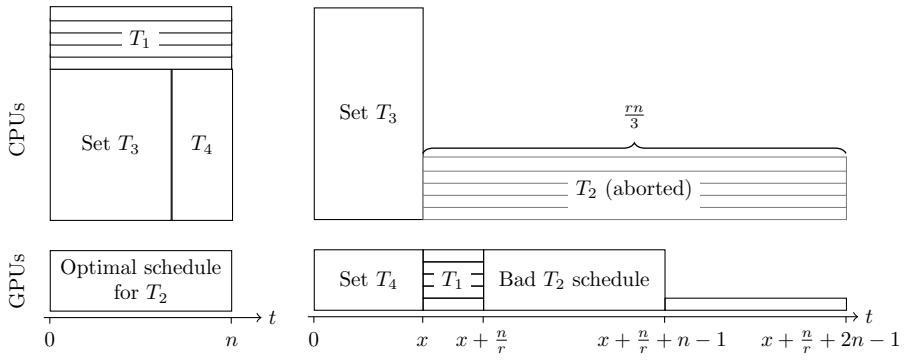


Fig. 4: 2 schedules for task set T_2 on $n = 6k$ homogeneous processors, tasks are labeled with processing times. Left one is an optimal schedule and right one is a possible list schedule.

the optimal makespan is a well known result. However, the interest of set T_2 is that the smallest execution time is $C^{\text{Opt}}_{\max}(T_2)/3$, what allows these tasks to have a large execution time on CPU in instance $\mathcal{I}$ (without having a too large acceleration factor).

Figure 5a shows an optimal schedule of length n for this instance: the tasks from set T_2 are scheduled optimally on the n GPUs, and the sets T_1, T_3 and T_4 are scheduled on the CPUs. Tasks T_3 and T_4 fit on the $m - n$ CPUs because the total work is $mx + nxr = x(m + nr) = (m - n)n$ by definition. On the other hand, Figure 5b shows a possible HeteroPrio schedule for $\mathcal{I}$. The tasks from set T_3 have the lowest acceleration factor and are scheduled on the CPUs, while tasks from T_4 are scheduled on the GPUs. All resources become available at time x. Tasks from set T_1 are scheduled on the n GPUs, and tasks from set T_2 are scheduled on m CPUs. At time $x + \frac{n}{r}$, the GPUs become available and start spoliating the tasks from set T_2. Since they all complete at the same time, the order in which they get spoliated can be arbitrary, and it can lead to the worst case behavior of Figure 4, where the task of length n is executed last. In this case, spoliating this task does not improve its completion time, and the resulting makespan for HeteroPrio on this instance is $C^{\text{HP}}_{\max}(I) = x + \frac{n}{r} + 2n - 1 = x + \frac{nr}{3}$ by definition of x. The approximation ratio on this instance is thus $C^{\text{HP}}_{\max}(I)/C^{\text{Opt}}_{\max}(I) = x/n + r/3$. When n becomes large, x/n tends towards 1, and r tends towards $3 + 2\sqrt{3}$. Hence, the ratio $C^{\text{HP}}_{\max}(I)/C^{\text{Opt}}_{\max}(I)$ tends towards $2 + 2/\sqrt{3}$, what ends the proof. □



(a) Optimal schedule (b) HeteroPrio when trying to spoliate the last task

Fig. 5: Optimal and HeteroPrio on Theorem 14 instance

6 Experimental evaluation

In this section, we propose an experimental evaluation of HeteroPrio on instances coming from the dense linear algebra library Chameleon [17]. We evaluate our algorithms in two contexts, (i) with independent tasks and (ii) with dependencies, which is closer to real-life settings and is ultimately the goal of the HeteroPrio algorithm. In this section, we use task graphs from Cholesky, QR and LU factorizations, which provide interesting insights on the behavior of the algorithms. The Chameleon library is built on top of the StarPU runtime, and implements tiled versions of many linear algebra kernels expressed as graphs of tasks. Before the execution, the processing times of the tasks are measured on both types of resources, which then allows StarPU schedulers to have a reliable prediction of each task's processing time. In this section, we use this data to build input instances for our algorithms, obtained on a machine with 20 CPU cores of two Haswell Intel® Xeon® E5-2680 processors and 4 Nvidia K40-M GPUs. We consider Cholesky, QR and LU factorizations with a tile size of 960, and a number of tiles N varying between 4 and 64.

We compare 3 algorithms from the literature : HeteroPrio, the well-known HEFT algorithm (designed for the general $R|prec|C_{\max}$ problem), and DualHP from [16] (specifically designed for CPU and GPU, with an approximation ratio of 2 for independent tasks). The DualHP algorithm works as follows : for a given guess λ on the makespan, it either returns a schedule of length 2λ, or ensures that $\lambda < C_{\max}^{Opt}$. To achieve this, any task with processing time more than λ on any resource is assigned to the other resource, and then all remaining tasks are assigned to the GPU by decreasing acceleration factor while the overall load is lower than $n\lambda$. If the remaining load on CPU is not more than $m\lambda$, the resulting schedule has makespan below 2λ. The best value of λ is then found by binary search.

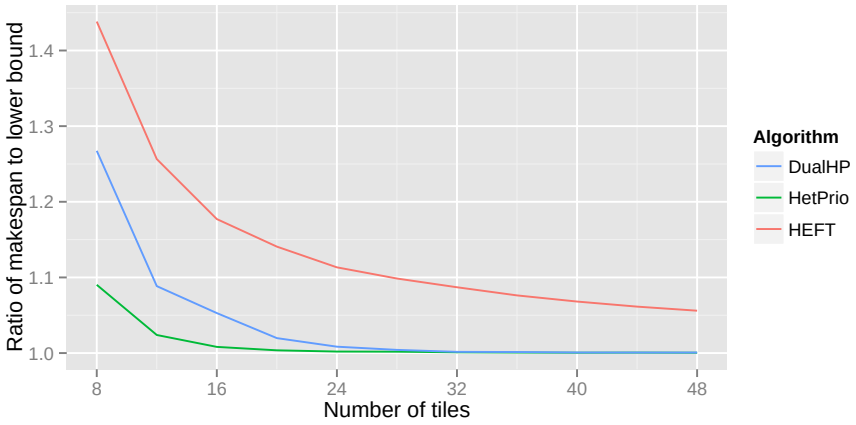


Fig. 6: Results for independent tasks

6.1 Independent Tasks

To obtain realistic instances with independent tasks, we have taken the actual measurements from tasks of each kernel (Cholesky, QR and LU) and considered these as independent tasks. For each instance, the performance of all three algorithms is compared to the area bound. Results are depicted in Figure 6, where the ratio to the area bound is given for different values of the number of tiles N. The results show that both HeteroPrio and DualHP achieve close to optimal performance when N is large, but HeteroPrio achieves better results for small values of N (below 20). This may be surprising, since the approximation ratio of DualHP is actually better than the one of HeteroPrio. On the other hand, HeteroPrio is primarily a list scheduling algorithm, that usually achieve good average case performance. In this case, it comes from the fact that DualHP tends to balance the *load* between the set of CPUs and the set of GPUs, but for such values of N, the task processing times on CPU are not negligible compared to the makespan. Thus, it happens that average loads are similar for both kinds of resources, but one CPU actually has significantly higher load than the others, what results in a larger makespan. HEFT, on the other hand, has rather poor performance because it does not take acceleration factor into account, and thus assigns tasks to GPUs that would be better suited to CPUs, and vice-versa.

6.2 Task Graphs

Both HeteroPrio and DualHP can be easily adapted to take dependencies into account, by applying at any instant the algorithm on the set of (currently) ready tasks. For DualHP, this implies recomputing the assignment of tasks to resources each time a task becomes ready, and also slightly modifying the algorithm to take into account the load of currently executing tasks. Since

HeteroPrio is a list algorithm, HeteroPrio rule can be used to assign a ready task to any idle resource. If no ready task is available for an idle resource, a spoliation attempt is made on currently running tasks.

When scheduling task graphs, a standard approach is to compute task priorities based on the dependencies. For homogeneous platforms, the most common priority scheme is to compute the *bottom-level* of each task, *i.e.* the maximum length of a path from this task to the exit task, where nodes of the graph are weighted with the execution time of the corresponding task. In the heterogeneous case, the priority scheme used in the standard HEFT algorithm is to set the weight of each node as the average execution time of the corresponding tasks on all resources. We will denote this scheme `avg`. A more optimistic view could be to set the weight of each node as the smallest execution time on all resources, hoping that the tasks will get executed on their favorite resource. We will denote this scheme `min`.

In both HeteroPrio and DualHP, these ranking schemes are used to break ties. In HeteroPrio, whenever two tasks have the same acceleration factor, the highest priority task is assigned first; furthermore, when several tasks can be spoliated for some resource, the highest priority candidate is selected. In DualHP, once the assignment of tasks to CPUs and GPUs is computed, tasks are sorted by highest priority first and processed in this order. For DualHP, we also consider another ranking scheme, `fifo`, in which no priority is computed and tasks are assigned in the order in which they became ready.

We thus consider a total of 7 algorithms: HeteroPrio, DualHP and HEFT with `min` and `avg` ranking schemes, and DualHP with `fifo` ranking scheme. We again consider three types of task graphs: Cholesky, QR and LU factorizations, with the number of tiles N varying from 4 to 64. For each task graph, the makespan with each algorithm is computed, and we consider the ratio to the lower bound obtained by adding dependency constraints to the area bound [12]. Results are depicted in Figure 7.

The first conclusion from these results is that scheduling DAGs corresponding to small or large values of N is relatively easy, and all algorithms achieve a performance close to the lower bound: with small values of N, the makespan is constrained by the critical path of the graph, and executing all tasks on GPU is the best option; when N is large, the available parallelism is large enough, and the runtime is dominated by the available work. The interesting part of the results is thus for the intermediate values of N, between 10 and 30 or 40 depending on the task graph. In these cases, the best results are always achieved by HeteroPrio, especially with the `min` ranking scheme, which is always within 30% of the (optimistic) lower bound. On the other hand, all other algorithms get significantly worse performance for at least one case.

To obtain a better insight on these results, let us further analyze the schedules produced by each algorithm by focusing on the following metrics: the amount of idle time on each type of resources (CPU and GPU)[1], and the ade-

[1] For fairness, any work made on an "aborted" task by HeteroPrio is also counted as idle time, so that all algorithms have the same amount of work to execute.

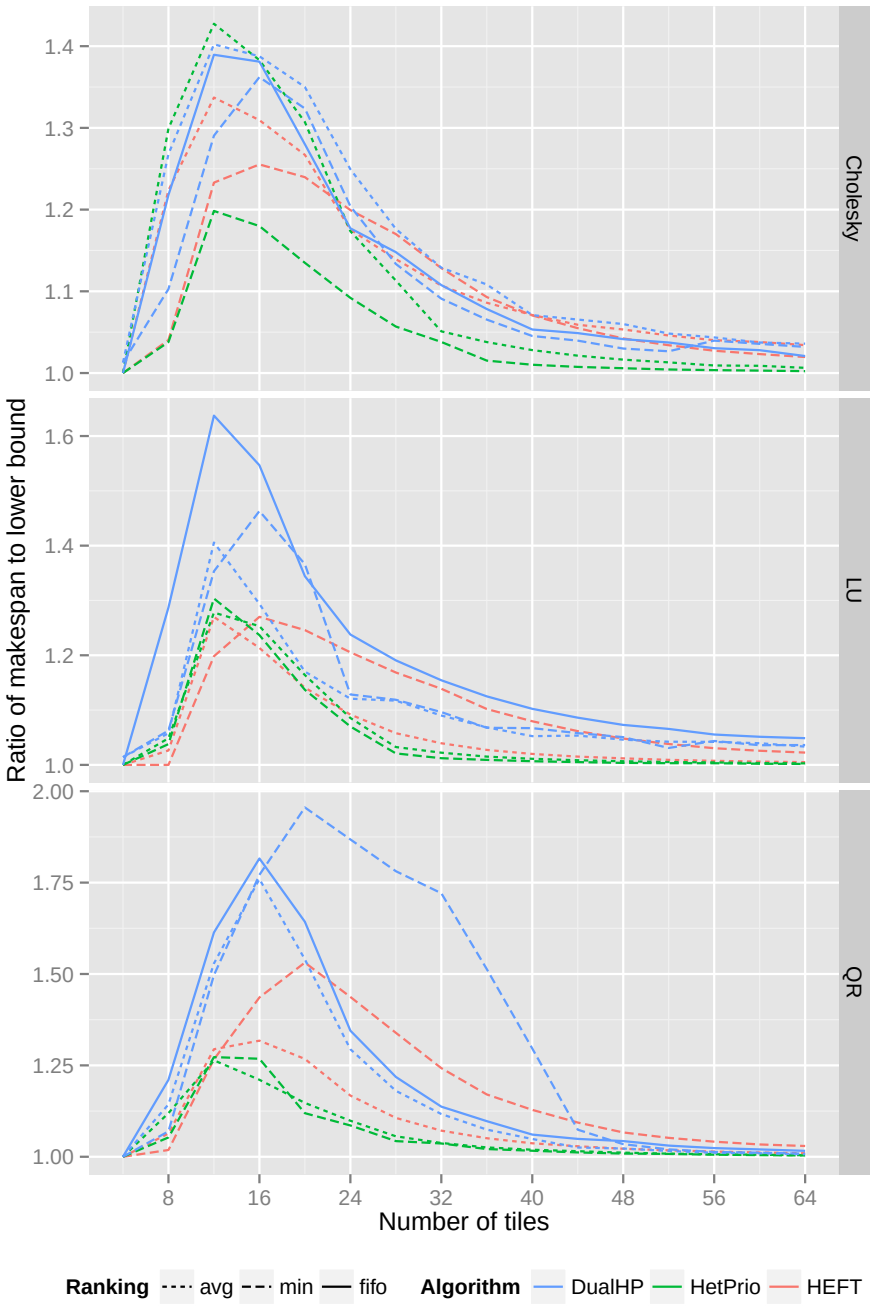


Fig. 7: Results for different DAGs

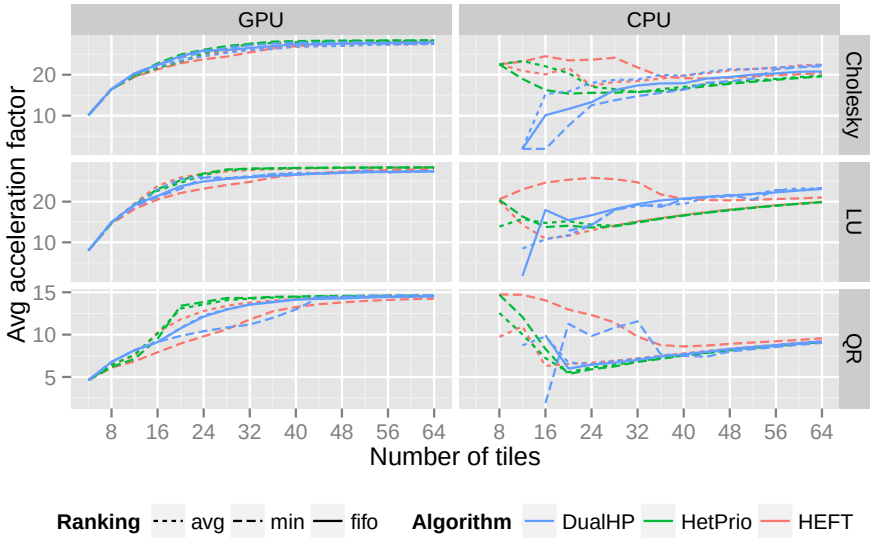


Fig. 8: Equivalent acceleration factors

quacy of task allocation (whether the tasks allocated to each resource is a good fit or not). To measure the adequacy of task allocation on a resource r, we define the acceleration factor $\mathcal{A}_r$ of the "equivalent task" made of all the tasks assigned to that resource: let J be the set of tasks assigned to r, $\mathcal{A}_r = \frac{\sum_{i \in J} p_i}{\sum_{i \in J} q_i}$. A schedule has a good adequacy of task allocation if $\mathcal{A}_{\text{GPU}}$ is high and $\mathcal{A}_{\text{CPU}}$ is low. The values of equivalent acceleration factors for both resources are shown on Figure 8. On Figure 9, the normalized idle time on each resource is depicted, which is the ratio of the idle time on a resource to the amount of that resource used in the lower bound solution.

On Figure 8, one can observe that there are significant differences in the acceleration factor of tasks assigned to the CPU between the different algorithms. In particular, HeteroPrio usually assigns to the CPU tasks with low acceleration factor (which is good), whereas HEFT usually has a higher acceleration factor on CPU. DualHP is somewhat in the middle, with a few exceptions in the case of LU when N is large. On the other hand, Figure 9 shows that HEFT and HeteroPrio are able to keep relatively low idle times in all cases, whereas DualHP induces very large idle time on the CPU. The reason for this is that optimizing locally the makespan for the currently available tasks makes the algorithm too conservative, especially at the beginning of the schedule where there are not many ready tasks, DualHP assigns all tasks on the GPU because assigning one on the CPU would induce a larger completion time. HeteroPrio however is able to find a good compromise by keeping the CPU busy with the tasks that are

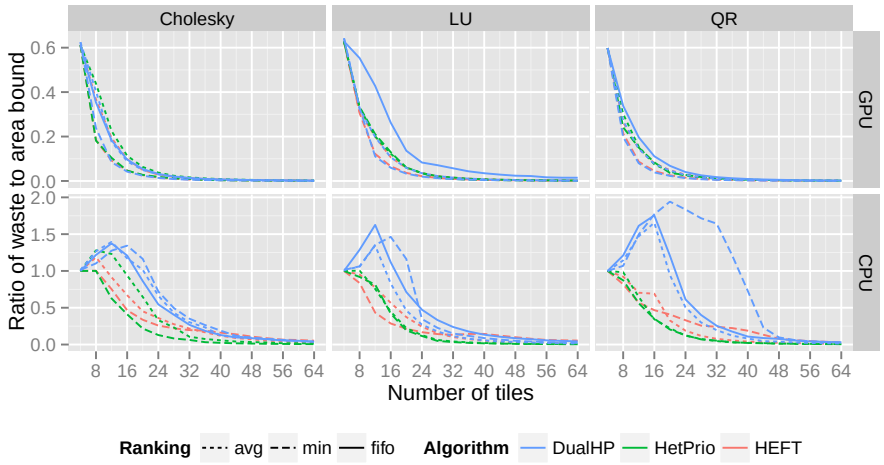


Fig. 9: Normalized idle time

not well suited for the GPU, and relies on the spoliation mechanism to ensure that bad decisions do not penalize the makespan.

7 Conclusion

In this paper, we consider HeteroPrio, a list-based algorithm for scheduling independent tasks on two types of unrelated resources. This scheduling problem has strong practical importance for the performance of task-based runtime systems, which are used nowadays to run high performance applications on nodes made of multicores and GPU accelerators. This algorithm has been proposed in a practical context, and we provide theoretical worst-case approximation proofs in several cases, including the most general, and we prove that our bounds are tight.

Furthermore, these algorithms can be extended to schedule tasks with precedence constraints, by iteratively scheduling the (independent) set of currently ready tasks. We show experimentally that in this context, HeteroPrio produces very efficient schedules, whose makespans are better than the state-of-the-art algorithms from the literature, and very close to the theoretical lower bounds. A practical implementation of HeteroPrio in the StarPU runtime system is currently under way.

References

[1] P. Brucker and S. Knust, "Complexity results for scheduling problems," Web document, URL: http://www2.informatik.uni-osnabrueck.de/knust/

class/.

[2] J. K. Lenstra, D. B. Shmoys, and É. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical programming*, 1990.

[3] R. Bleuse, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling Independent Tasks on Multi-cores with GPU Accelerators," *Concurr. Comput. : Pract. Exper.*, vol. 27, no. 6, pp. 1625–1638, Apr. 2015.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.

[5] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[6] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. Van de Geijn, "SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks," in *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, p. 123–132.

[7] A. YarKhan, J. Kurzak, and J. Dongarra, *QUARK Users' Guide: QUeueing And Runtime for Kernels*, UTK ICL, 2011.

[8] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations," in *Euro-Par (2)*, 2010, pp. 235–246.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Computing in Science and Engineering*, 2013.

[10] V. Bonifaci and A. Wiese, "Scheduling unrelated machines of few different types," *CoRR*, vol. abs/1205.0974, 2012. [Online]. Available: http://arxiv.org/abs/1205.0974

[11] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[12] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Are Static Schedules so Bad? A Case Study on Cholesky Factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, 2016, pp. 1021–1030.

[13] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for heterogeneous architectures," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 9, Jun. 2016.

[14] E. V. Shchepin and N. Vakhania, "An optimal rounding gives a better approximation for scheduling unrelated machines," *Operations Research Letters*, 2005.

[15] C. Imreh, "Scheduling problems on two sets of identical machines," *Computing*, vol. 70, no. 4, pp. 277–294, 2003.

[16] R. Bleuse, T. Gautier, J. V. F. Lima, G. Mounié, and D. Trystram, *Scheduling Data Flow Program in XKaapi: A New Affinity Based Algorithm for Heterogeneous Architectures.* Cham: Springer International Publishing, 2014, pp. 560–571.

[17] "Chameleon, A dense linear algebra software for heterogeneous architectures," 2014. [Online]. Available: https://project.inria.fr/chameleon