Scheduling independent tasks on multi-cores with GPU accelerators

Raphaël Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, Denis Trystram

SCHEDULING INDEPENDENT TASKS ON MULTI-CORES WITH GPU ACCELERATORS

RAPHAEL BLEUSE[1], SAFIA KEDAD-SIDHOUM[2], FLORENCE MONNA[1,2], GRÉGORY MOUNIÉ[2], AND DENIS TRYSTRAM[2,3]

Abstract. More and more computers use hybrid architectures combining multi-core processors and hardware accelerators like GPUs (Graphics Processing Units). We present in this paper a new method for scheduling efficiently parallel applications with m CPUs and k GPUs, where each task of the application can be processed either on a core (CPU) or on a GPU. The objective is to minimize the maximum completion time (makespan). The corresponding scheduling problem is NP-hard, we propose an efficient approximation algorithm which achieves an approximation ratio of $\frac{4}{3} + \frac{1}{3k}$. We first detail and analyze the method, based on a dual approximation scheme, that uses dynamic programming to balance evenly the load between the heterogeneous resources. Then, we present a faster approximation algorithm for a special case of the previous problem, where all the tasks are accelerated when affected to GPU, with a performance guarantee of $\frac{3}{2}$ for any number of GPUs. We run some simulations based on realistic benchmarks and compare the solutions obtained by a relaxed version of the generic method to the one provided by a classical scheduling algorithm (HEFT). Finally, we present an implementation of the 4/3-approximation and its relaxed version on a classical linear algebra kernel into the scheduler of the xKaapi runtime system.

1. Introduction

Most of the computing systems available today include parallel multi-core chips sharing a large memory with additional hardware accelerators [1]. There is an increasing complexity within the internal nodes of such parallel systems, mainly due to the heterogeneity of the computational resources. In order to take advantage of the benefits offered by heterogeneity in terms of performance, effective and automatic management of the hybrid resources will be more and more important for running any applications. These new hybrid architectures have given rise to new scheduling problems consisting in allocating and sequencing the computations on the different resources such that a given objective is optimized.

The main challenge is to create adequate generic methods and software tools that fulfill the requirements for optimizing the performances. In the field of High Performance Computing (HPC), one of the most studied optimization problem is the minimization of the maximum completion time (makespan) of the schedule, which is the objective addressed in this paper. Some Polynomial Time Approximation Schemes (PTAS) exist for problems of minimizing the makespan on heterogeneous processors [2, 3], but their running times make them impractical for solving scheduling problems on actual computing platforms.

[1] Univ. Grenoble-Alpes, LIG, F-38000 Grenoble, France., CNRS, LIG, F-38000 Grenoble, France, Inria

[2] Sorbonne Universités, UPMC Univ.Paris 06, UMR 7606, LIP6, F-75005, Paris, France

[3] Institut Universitaire de France

Key words and phrases. Scheduling; Approximation algorithms; Parallel heterogeneous systems.

This work was partially supported by the French Ministry of Defense (DGA)..

In the field of parallel processing, there exist a huge number of papers dealing with implementations of *ad hoc* algorithms using GPUs or hybrid architectures. They expand over several aspects of parallelism from operating system, runtime, application implementation or languages. However, only few of them focus on the intermediate problem of scheduling on hybrid platforms [4]. Most of the works in the literature consist in studying the gains and performances of parallel implementation of some specific numerical kernels [5, 6], specific applications like multiple alignments of biological sequences [7], or molecular dynamics [8]. The existing scheduling algorithms and tools are usually not well-suited for general purpose applications since the internal hardware organization of a GPU highly differs from a CPU and thus, the GPU should be considered as a new type of resource in order to determine efficient approaches. Scheduling is usually done on a case by case basis and often offers good performances, however, it lacks high-level mechanisms that provide transparent and efficient schedules for any application. Some actual runtime systems include the basic mechanisms for developing scheduling algorithms like OMPSS [9], StarPU [10] or xKaapi [11]. Several scheduling algorithms have been implemented on top of these systems. An online algorithm with a performance guarantee [12] has recently been developed for CPU-GPU platforms, but there is no performance guarantee for any offline problem on these systems.

Our objective within this work is to propose a new method for scheduling independent tasks on hybrid CPU-GPU architectures designed for HPC. The considered input is a set of independent sequential tasks whose execution times are known. This hypothesis is realistic, since some computing systems such as StarPU have a module which estimates the execution times at compile time. The method that we propose in this work determines the allocation and schedule of the tasks to the computing units, CPUs and GPUs. We present and analyze in detail this methodology for the case of scheduling n sequential tasks on m cores (CPUs) and k GPUs. This leads to an efficient approximation algorithm which achieves a ratio $\frac{4}{3} + \frac{1}{3k} + \epsilon$ using dual approximation [13] with a dynamic programming scheme. The computational cost of this algorithm is in $\mathcal{O}\left(n^2k^3m^2\right)$ per step of dual approximation. We also present an approximation algorithm for the more specific problem of scheduling a set of taskson the same hybrid platform that are all accelerated when processed on a GPU, although they can have different accelerations. This algorithm also relies on the dual approximation technique and achieves a $\frac{3}{2}$ performance ratio for a time complexity in $\mathcal{O}\left(mn \log n\right)$. As the method with a $\frac{4}{3}$ approximation ratio is rather costly, in the perspective of an integration into actual runtime systems, we derive a relaxed algorithm and compare it experimentally with one of the most popular algorithms (HEFT [14]), as well as with the approximation algorithm presented for the case where all the tasks are accelerated on GPUs.

The outline of the paper is as follows. A formal description of the scheduling problem with k GPUs is provided in Section 2, and some related works are presented. We propose in Section 3 a new approach for solving the problem. We propose in Section 4 an algorithm for the scheduling problem with k GPUs where all tasks are accelerated when processed on GPU. We report the results of experiments in Section 5 where a relaxed version of the method presented in Section 3 and the algorithm from Section 4 are compared to the classical HEFT algorithm on simulations built from realistic workloads. The experimental analysis shows that the proposed methods have a more stable behavior than HEFT for a similar performance. Then, an implementation of both the generic method and its relaxed version on a real run-time system have been realized and tested on a classical Linear Algebra kernel. Finally, a synthesis is provided in Section 6, which opens some perspectives.

Some of this work was presented at the HeteroPar 2013 conference [15], but the approximation algorithm for the specific problem of scheduling a set of tasks that are all accelerated when processed on a GPU is an entirely new contribution, and the experiments were extended in comparison to the original paper, where only partial simulations for the main algorithm were presented. Practically, the cost of the $\frac{4}{3}$ algorithm is too expensive since the number of independent tasks is too low in the considered kernel. The relaxed 2-approximation is shown to be as good as HEFT, but provides a more stable behavior and a lower volume of communications.

2. Problem Definition and Related Works

We consider a multi-core parallel platform composed of m identical CPUs and k identical GPUs. The m CPUs are considered independent from the GPUs that are commanded by some extra driving CPUs, not mentioned here because they do not execute any task. An application is composed of n independent tasks denoted by $T_1, \ldots, T_n$. The tasks are considered sequential, meaning that they only are processed on one processor of any type, CPU or GPU. Each of these tasks has two processing times depending on which type of processor it is allocated to. The processing time of task T_j is denoted by $\overline{p_j}$ if T_j is processed on a CPU and $\underline{p_j}$ if it is processed on a GPU. The acceleration factor of task T_j will be given by the ratio $\frac{\overline{p_j}}{\underline{p_j}}$. We assume that both processing times of a task are known in advance as it is commonly admitted. An accurate estimation can be obtained at compile time for regular numerical applications in HPC. The objective here is to minimize the makespan $C_{\max}$ of the whole schedule. It is defined as the maximum completion time of the last finishing task on both CPUs and GPUs, $C_{max} = \max(C_{max}^{CPU}, C_{max}^{GPU})$. The problem will be denoted as $(Pm, Pk) \mid\mid C_{\max}$.

If both processing times are equal ($\overline{p_j} = \underline{p_j}$) for all $j = 1, \ldots, n$, the problem $(P1, P1) \mid\mid C_{\max}$ is equivalent to the classical $P2 \mid\mid C_{\max}$ problem, which is NP-hard [16]. Thus, the problem of scheduling with GPUs is also NP-hard. Our objective is to propose efficient approximation algorithms with a performance guarantee. Recall that for a given problem, the approximation ratio ρ_A of an algorithm A solving this problem is defined as the maximum over all the instances I of the ratio $\frac{f(I)}{f^*(I)}$ where f is any minimization objective and f^* is its optimal value [13].

The problem considered here is a new problem, harder than the classical scheduling problem on uniform machines $Q \mid\mid C_{max}$, but better results can be expected if the problem is considered on its own rather than as an unrelated scheduling problem $R \mid\mid C_{\max}$ [17]. Lenstra *et al.* [18] proposed a PTAS for the problem $R \mid\mid C_{\max}$ with running time bounded by the product of $(n+1)^{m/\epsilon}$ and a polynomial of the input size. Let us notice that if m is not fixed, then the algorithm is not fully polynomial. The authors also proved that unless $P = NP$, there is no polynomial-time approximation algorithm for $R \mid\mid C_{\max}$ with an approximation factor less than $\frac{3}{2}$ and they presented a 2-approximation algorithm. This algorithm is based on rounding an optimal solution of the preemptive version of the problem obtained by an integer linear program. Shmoys and Tardos [19] generalized this technique to obtain the same approximation factor for the generalized assignment problem. Furthermore, they generalized the rounding technique to hold for any fractional solution. Recently, Shchepin and Vakhania [20] introduced a new rounding technique which yields an improved approximation factor of $2 - \frac{1}{m}$ for $R \mid\mid C_{\max}$ for a similar time complexity as [18]. According to our knowledge, this is so far the best approximation result for this problem. However, the prohibitive computational cost of these algorithms prevents their usage on actual computing platforms.

It is worth noticing that if all the tasks of the problem have the same acceleration on the GPUs, the problem reduces to the classical $Q \mid\mid C_{\max}$ problem, with two machine speeds. For $Qm \mid\mid C_{\max}$, Friesen [21] proved that the approximation ratio of the well-known Longest Processing Time (LPT) scheduling policy satisfies $1.52 \leqslant C_{\max}^{LPT}/C_{\max}^{*} \leqslant 5/3$. The first PTAS for $Q \mid\mid C_{\max}$ was given by Hochbaum and Shmoys [2]. The overall running time of the algorithm is $\mathcal{O}((\log m + \log(3/\epsilon))(m/\epsilon)(n/\epsilon)1/\epsilon)$. However, these solving methods would only work for specific instances of the problem of scheduling on hybrid platforms, where the acceleration factors $\frac{\overline{p_j}}{\underline{p_j}}$ would be equal to a constant, which is not the case in practice.

Another direction is to consider the problem of scheduling on unrelated machines of few different types. Indeed, the $R \mid\mid C_{\max}$ reference problem can be refined to better fit the constraints of the hybrid platforms. Bonifaci and Wiese [3] presented a PTAS to solve a scheduling problem with unrelated machines of few different types. The tools used in their solving method are somewhat similar to the ones used for solving $R \mid\mid C_{\max}$, and the rounding phases of the algorithm require a significant amount of time, raising the time complexity of the algorithm to an impractical level, even when only two types of machines are considered, as it is for $(Pm, Pk) \mid\mid C_{\max}$. There is a need to consider other algorithms than these PTAS to design algorithms that could be implemented on actual platforms. A PTAS with a reasonable time complexity has been developed for the online version of the problem of the assignment of sporadic tasks on hybrid platforms [22]. However, an offline version of the problem with non-periodic tasks has not been studied and the algorithm cannot be trivially extended to the problem $(Pm, Pk) \mid\mid C_{\max}$. On another side, Imreh [23] presented different greedy algorithms for the problem of scheduling on two sets of identical machines, with varying approximation ratios including $2 + \frac{m-1}{k}$ and $4 - \frac{2}{m}$ (for m CPUs and k GPUs). These algorithms are fast enough for being implemented in modern platforms, nevertheless the approximation ratios of these algorithms are quite high since usually the number of GPUs is much lower than the number of CPUs.

From a practical perspective, the scheduling strategy is a key point for the performance of an application. Tuning scheduling algorithms for a specific case (problem and computer architecture) is common. Fast heuristics without performance guarantee are often used on computing platforms, time efficiency being the crucial factor. However, simple strategies are not sufficient to guarantee the performance for more general cases potentially far from the specific one. The performance portability is difficult to achieve when the number of CPUs and GPUs varies or when the speedup of the various parts of the application is evolving during the execution.

Our objective within this work is to build a bridge between purely theoretical algorithms with good performance guarantees and practical low cost heuristics. Thus, we propose a tradeoff solution with a provable performance guarantee and a reasonable time complexity.

3. A New Algorithm

3.1. **Rationale of the solving method.** The proposed algorithm is based on the dual approximation technique [13]. A g-dual approximation algorithm for a generic problem takes a real number λ (guess) as an input, assumes that there exists a schedule of length at most λ and either delivers a schedule of makespan at most $g\lambda$, or answers correctly that there exists no schedule of length at most λ. A binary search is used to try different guesses to approach the optimal makespan as follows: we first take an initial lower bound B_{min} and an initial upper bound B_{max} of the

optimal makespan. We start by solving the problem with a λ equal to the average of these two bounds and then the bounds are updated as follows:

- If the algorithm returns "NO", then λ becomes the new lower bound.
- If the algorithm returns a schedule of makespan at most $g\lambda$, then there exists a schedule of makespan at most λ and λ becomes the new upper bound.

The number of iterations of the binary search is bounded by $\log_2(B_{max} - B_{min})$. Hence, a g-dual approximation algorithm can be converted, by bisection search, in a $g(1+\epsilon)$-approximation algorithm with a similar running time.

We target $g = \frac{4}{3} + \frac{1}{3k}$. Let λ be the current real number input (guess) for the dual approximation. The key point is to show how it is possible to build a schedule of length at most $\frac{4\lambda}{3} + \frac{\lambda}{3k}$, starting from the assumption that there exists a schedule of length lower than λ.

The idea is to partition the set of tasks on the CPUs into two sets, each consisting in two shelves: a first set with a shelf of length λ and the other of length $\frac{\lambda}{3}$, and a second set with two shelves of length $\frac{2\lambda}{3}$. The partition ensures that the makespan on the CPUs is lower than $\frac{4\lambda}{3}$. The same partition can be applied to the set of tasks on the GPUs, with the smallest shelf of length $\frac{\lambda}{3} + \frac{\lambda}{3k}$ instead of $\frac{\lambda}{3}$. Since the tasks are independent, the scheduling strategy is straightforward when the assignment of the tasks has been determined and yields directly a solution of length at most $\frac{4\lambda}{3}$. The main problem is to assign the tasks in each shelf on the CPUs or on the GPUs in order to obtain a feasible solution. This will be done using dynamic programming. The main steps are summarized in the following algorithmic scheme:

(1) Compute the guess $\lambda = \frac{B_{min}+B_{max}}{2}$ where B_{min} (resp. B_{max}) is a lower (resp. upper) bound of the optimal makespan.
(2) Search for an allotment of the tasks such that:
   - the total load (work) on CPUs is at most $m\lambda$,
   - the total load (work) on GPUs is at most $k\lambda$,
   - the tasks alloted to the CPUs (resp. GPUs) whose processing time is strictly greater than $\frac{2\lambda}{3}$ occupy a maximum number of CPUs (resp. GPUs) denoted by μ (resp. κ).
   - the tasks alloted to the CPUs (resp. GPUs) whose processing time is strictly greater than $\frac{\lambda}{3}$ and lower than $\frac{2\lambda}{3}$ can be assigned two by two to a maximum number of CPUs (resp. GPUs) denoted by $\mu'/2$ (resp.$\kappa'/2$).
     The total number of CPUs (resp. GPUs) must not exceed m (resp. k), i.e. $\mu + \mu'/2 \leq m$ (resp. $\kappa + \kappa'/2 \leq k$),
   - the tasks assigned to the CPUs (resp. GPUs) with processing time lower that $\frac{\lambda}{3}$ can be scheduled such that the induced makespan on the CPUs (resp. GPUs) will be at most equal to $\frac{4\lambda}{3}$ (resp. $\frac{4\lambda}{3} + \frac{\lambda}{3k}$).
(3) If such an allotment does not exist, adjust the bound B_{min} to λ and restart the process (Step 1).
(4) If such an allotment exists, build the corresponding schedule with sets of shelves such that the makespan is lower than $\frac{4}{3}\lambda$, adjust the bound B_{max} to λ and restart the process.

3.2. **Structure of an Optimal Schedule of length at most λ.** We introduce an allocation function $\pi(j)$ of a task T_j which corresponds to the processor where the task is processed. The set $\mathcal{C}$ (resp. $\mathcal{G}$) is the set of all the CPUs (resp. GPUs). Therefore, if a task T_j is assigned to a CPU, we can write $\pi(j) \in \mathcal{C}$. We define W_C as the computational area of the CPUs on the Gantt chart representation of a schedule, i.e. the sum of all the processing times of the tasks allocated to the

CPUs: $W_C = \sum_{j\,/\,\pi(j)\in\mathcal{C}} \overline{p_j}$. This corresponds to the computational load of all the CPUs.

To take advantage of the dual approximation paradigm, we have to make explicit the consequences of the assumption that there exists a schedule of length at most λ. We state below some straightforward properties of such a schedule. They should give the insight for the construction of the solution.

Property 1. *In an optimal solution, the execution time of each task is at most λ, and the computational area on the CPUs (resp. GPUs) is at most $m\lambda$ (resp. $k\lambda$).*

Property 2. *In an optimal solution, if there exist two tasks executed on the same processor such that one of these tasks has an execution time greater than $\frac{2\lambda}{3}$, then the other one has an execution time lower than $\frac{\lambda}{3}$.*

Property 3. *Two tasks with processing times on CPU (resp. GPU) greater than $\frac{\lambda}{3}$ and lower than $\frac{2\lambda}{3}$ can be executed successively on the same CPU (resp. GPU) within a time at most $\frac{4\lambda}{3}$.*

The basic idea of the solution that we propose comes from the analysis of the shape of an optimal schedule. From Property 2, the tasks whose execution times on CPU (resp. on GPU) are strictly greater than $\frac{2\lambda}{3}$ do not use more than m CPUs (resp. k GPUs), and hence can be executed concurrently in the first set in a shelf denoted by S_1 (resp. S_5). We denote by μ (resp. κ) the number of CPUs (resp. GPUs) executing these tasks. These shelves are represented for the CPUs in Figure 1.

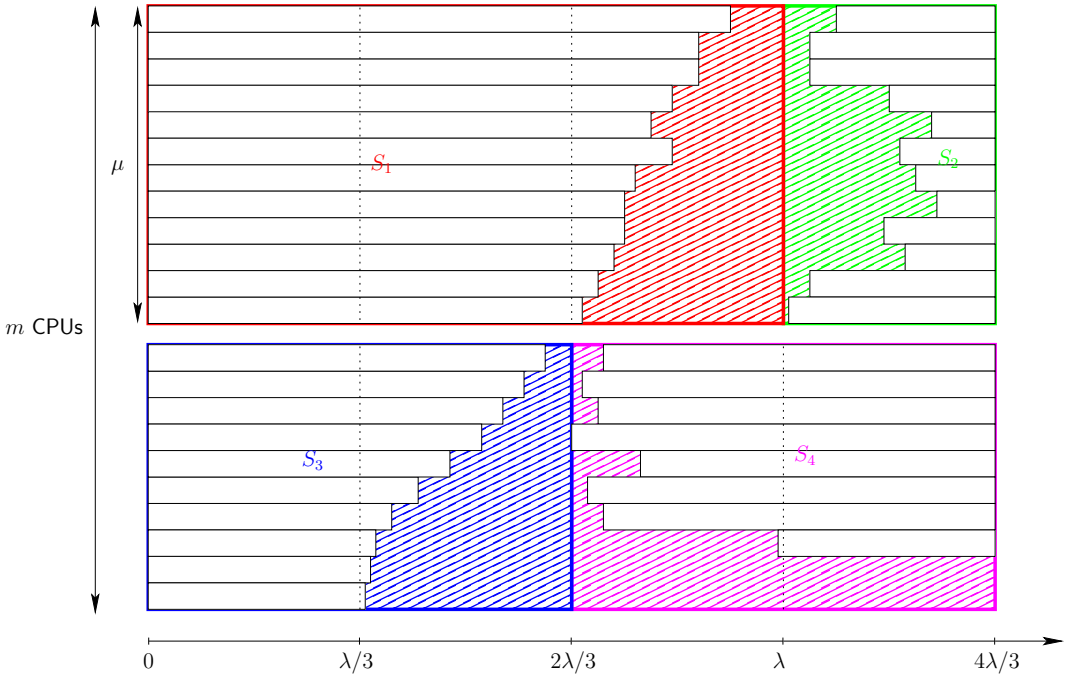

FIGURE 1. Partitioning the set of tasks on the CPUs into two sets of two shelves, the first one occupying μ CPUs, the second $m - \mu$ CPUs.

The tasks whose execution times are lower than $\frac{2\lambda}{3}$ and strictly greater than $\frac{\lambda}{3}$ on CPU (resp. on GPU) cannot be executed on the μ CPUs (resp. κ GPUs)

occupied by S_1 (resp. S_5) from Property 1. Moreover, from Property 3, $2(m-\mu)$ (resp. $2(k-\kappa)$) of these tasks on CPU (resp. GPU) can be executed in time at most $\frac{4\lambda}{3}$ on the remaining $(m-\mu)$ CPUs (resp. $(k-\kappa)$ GPUs) in the second set and fill two shelves S_3 and S_4 (resp. S_7 and S_8) of equal length $\frac{2\lambda}{3}$.

The tasks remaining to be assigned to the CPUs (resp. GPUs) have a processing time lower than $\frac{\lambda}{3}$. The μ (resp. κ) longest remaining tasks are assigned to the first set on the CPUs (resp. GPUs) in another shelf denoted by S_2 (resp. S_6). The length of S_2 is $\frac{\lambda}{3}$ and S_6 has a length of $\frac{\lambda}{3}+\frac{\lambda}{3k}$.

W_L (resp. W_R) will denote the computational area on the CPUs (resp. GPUs) remaining idle after this allocation in the schedule of length $\frac{4\lambda}{3}+\frac{\lambda}{3k}$. W_L corresponds to the stripped area in Figure 1. Regarding the question of how the remaining tasks fit in the constructed schedule, we state the following lemma:

Lemma 1. *The tasks remaining to be assigned on the CPUs after the construction of S_1, S_2, S_3, S_4 fit in the remaining free computational space W_L between these shelves.*

Proof. The tasks remaining to be assigned after the construction of $S_1, \ldots, S_4$ all have a processing time lower than $\frac{\lambda}{3}$ by construction and they necessarily fit into the remaining computational space W_L, otherwise the schedule would not satisfy Property 1. The following algorithm can be used to schedule these tasks:

- Consider the remaining tasks ordered by decreasing order of processing time on CPU $T_1, \ldots, T_f$, f being the total number of tasks remaining to be allocated.
- At each step i, $i = 1, \ldots, f$, allocate task T_i to the least loaded processor, at the latest possible date. Update its load.

At each step, the least loaded processor has a load at most λ; otherwise it would contradict the fact that the total work area of the tasks is bounded by $m\lambda$ (according to Property 1). Hence, the idle time interval on the least loaded CPU has a length at least equal to $\frac{\lambda}{3}$ and can contain the task T_i, which proves the correctness of the scheduling algorithm. □

The question of how the remaining tasks to be assigned on GPUs fit in the constructed schedule will be addressed later in the paper (see Lemma 2) .

3.3. **Partitioning the Tasks into Shelves.** In this section, we detail how to fill the shelves (see Figure 1) on the CPUs (resp. GPUs) by specifying an initial assignment of the tasks to the processors.

In order to obtain a 2-sets and 4-shelves schedule on the CPUs (resp. GPUs), we look for an assignment satisfying the following constraints:

- (C_1) The total computational area W_C on the CPUs is at most $m\lambda$.
- (C_2) The set $\mathcal{T}_1$ of tasks on the CPUs with an execution time strictly greater than $\frac{2\lambda}{3}$ in the allotment, to be scheduled in S_1, uses a total of at most m processors. We still denote by μ the number of processors they use.
- (C_3) The set $\mathcal{T}_2$ of tasks on the CPUs with an execution time lower than $\frac{2\lambda}{3}$ and strictly greater than $\frac{\lambda}{3}$ in the allotment, to be scheduled in S_3 or S_4, uses a total of at most $2(m-\mu)$ processors.
- (C_4) The total computational area on the GPUs is at most $k\lambda$.
- (C_5) The set $\mathcal{T}_3$ of tasks on the GPUs with an execution time strictly greater than $\frac{2\lambda}{3}$ in the allotment, to be scheduled in S_5, uses a total of at most k processors. We still denote by κ the number of processors they use.
- (C_6) The set $\mathcal{T}_4$ of tasks on the GPUs with an execution time lower than $\frac{2\lambda}{3}$ and strictly greater than $\frac{\lambda}{3}$ in the allotment, to be scheduled in S_7 or S_8, uses a total of at most $2(k-\kappa)$ processors.

Let us notice that if constraints (C_3) and (C_6) are satisfied, then constraints (C_2) and (C_5) will also be satisfied. Hence, constraints (C_2) and (C_5) are relaxed.

We define for each task T_j a binary variable x_j such that $x_j = 1$ if T_j is assigned to a CPU or 0 if T_j is assigned to a GPU. Determining if an allotment satisfying (C_1), (C_3), (C_4) and (C_6) exists reduces to solving a three-dimensional knapsack minimization problem that can be formulated as follows:

$$W_C^* = \min \sum_{j=1}^{n} \overline{p_j} x_j \tag{1}$$

$$\text{s.t. } \frac{1}{2} \sum_{2\lambda/3 \geqslant \overline{p_j} > \lambda/3} x_j + \sum_{\overline{p_j} > 2\lambda/3} x_j \leqslant m \tag{2}$$

$$\frac{1}{2} \sum_{2\lambda/3 \geqslant \underline{p_j} > \lambda/3} (1 - x_j) + \sum_{\underline{p_j} > 2\lambda/3} (1 - x_j) \leqslant k \tag{3}$$

$$\sum_{j=1}^{n} \underline{p_j} (1 - x_j) \leqslant k\lambda \tag{4}$$

$$x_j \in \{0, 1\} \tag{5}$$

Equation (1) represents the minimal workload on all the CPUs. Constraint (2) (resp. Constraint (3)) imposes that no more than m (resp. k) tasks can be executed on the CPUs (resp. GPUs) with a processing time greater than $\frac{2\lambda}{3}$, we note $\mu = \sum_{\overline{p_j} > 2\lambda/3} x_j$ (resp. $\kappa = \sum_{\underline{p_j} > 2\lambda/3} (1 - x_j)$) their number; and that there cannot be more than $2(m - \mu)$ (resp. $2(k - \kappa)$) tasks on the CPUs (resp. GPUs) with a processing time lower than $\frac{2\lambda}{3}$ and greater than $\frac{\lambda}{3}$ (cf. Constraint (C_3) (resp. (C_6)))). Constraint (4) imposes an upper bound on the computational area on the GPUs which is $k\lambda$ (cf. (C_4)).

We propose a dynamic programming algorithm in $\mathcal{O}\left(n^2 m^2 k^3\right)$ to solve the knapsack problem. For this purpose, we first discretize the processing times of the tasks on the GPUs. We introduce $\nu_j = \left\lfloor \frac{\underline{p_j}}{\lambda/(3n)} \right\rfloor$ to represent the number of integer time intervals of length $\frac{\lambda}{3n}$ required for a task T_j if it is executed on the GPUs, as shown in Figure 2. $N = \sum_{\pi(j) \in \mathcal{G}} \nu_j$ denotes the total integer number of these intervals on the GPUs. We thus define the error on the processing time of each task $\epsilon_j = \underline{p_j} - \nu_j \frac{\lambda}{3n}$ induced by this time discretization.

This result allows us to consider only N states in the dynamic programming regarding the workload on the GPUs. The error ϵ_j on each task is at most $\frac{\lambda}{3n}$, so if all the tasks were assigned to one of the GPU, we would have underestimated the processing time on this GPU by at most $n\frac{\lambda}{3n} = \frac{\lambda}{3}$. Constraint (4) becomes:

$$N = \sum_{\pi(j) \in \mathcal{G}} \nu_j \leqslant 3kn \tag{6}$$

The approximated computational area of the GPUs is at most $k\lambda$ and thus, the full computational area on GPU remains lower than $k\lambda + \frac{\lambda}{3}$. This allows us to answer the question of how the remaining tasks to be assigned on GPUs fit in the constructed schedule:

Lemma 2. *The tasks remaining to be assigned on the GPUs after the construction of S_5, S_6, S_7, S_8 fit in the remaining free computational space W_R between these shelves.*

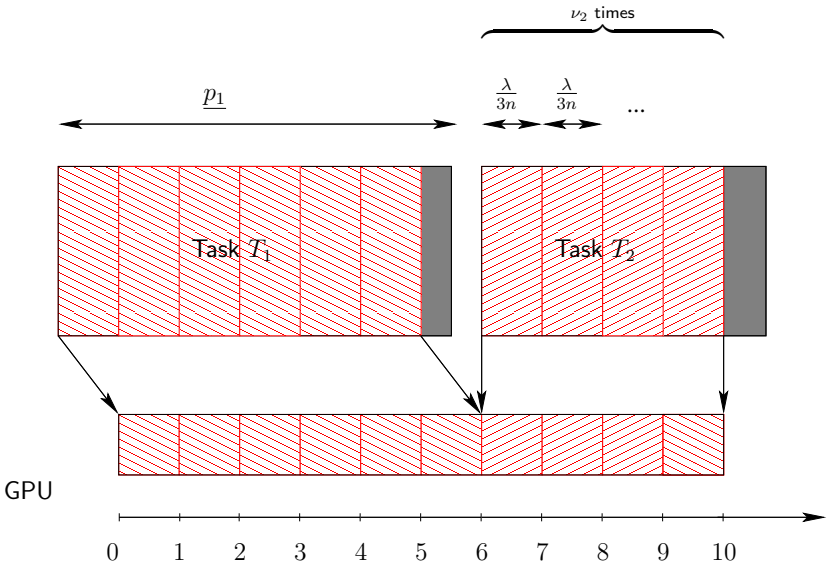

FIGURE 2. Rounded allocation of two tasks T_1 with $\underline{p_1} = 6.5$ and T_2 with $\underline{p_2} = 4.7$ on a GPU

Proof. The proof is similar to the one of Lemma 1. If we modify the starting time of the tasks of S_6, currently λ, so that all the working processors complete their tasks at $\frac{4\lambda}{3} + \frac{\lambda}{3k}$, creating an idle time interval between the end of S_5 and the starting time of S_6, the load of a GPU is equal to $\frac{4\lambda}{3} + \frac{\lambda}{3k}$ minus the length of the idle time interval.

With the same algorithm as for Lemma 1, the only problem that may occur is if a task T_i remaining to be allocated cannot be completed before the starting time of the tasks of S_6. But at each step, the least loaded processor has a load at most $\lambda + \frac{\lambda}{3k}$ since the total work area of the tasks is bounded by $k\left(\lambda + \frac{\lambda}{3k}\right)$. Hence, the idle time interval on the least loaded GPU has a length at least $\frac{\lambda}{3}$ and can contain the task T_i. □

We define $W_C(j, \mu, \mu', \kappa, \kappa', N)$ as the minimum sum of all the processing times of the tasks on the CPUs when the first j tasks are considered, with among the tasks on the CPUs (resp. GPUs), μ (resp. κ) of them having processing times greater than $\frac{2\lambda}{3}$ and μ' (resp. κ') with $\frac{\lambda}{3} < \overline{p_j}$ (resp. $\underline{p_j}$) $\leqslant \frac{2\lambda}{3}$, and where N time intervals are occupied on the GPUs.

We use a dynamic programming algorithm to compute the value of $W_C(j, \mu, \mu', \kappa, \kappa', N)$ with the values of W_C with $j-1$ tasks considered that were previously computed. The optimal value of the computational area W_C on the CPUs will be given by

$$W_C^* = \min_{0\leqslant\mu\leqslant m,\ 0\leqslant\mu'\leqslant 2(m-\mu),\ 0\leqslant\kappa\leqslant k,\ 0\leqslant\kappa'\leqslant 2(k-\kappa),\ 0\leqslant N\leqslant 3kn} W_C\left(n, \mu, \mu', \kappa, \kappa', N\right)$$

If W_C^* is greater than $m\lambda$, then there exists no solution with a makespan at most λ, and the algorithm answers "NO" in the dual approximation framework. Otherwise, the guess λ is large enough, we construct a feasible solution with a makespan at most $\frac{4\lambda}{3} + \frac{\lambda}{3k}$, with the shelves and the corresponding μ, μ', κ, κ' and N values.

The dynamic programming algorithm represents one step of the dual-approximation algorithm, with a fixed guess λ. A binary search is then used to try different guesses to approach the optimal makespan as explained in Section 3.1.

Cost Analysis. Solving the dynamic programming algorithm for a fixed value of λ requires to consider $\mathcal{O}\left(n^2m^2k^3\right)$ states, since $1 \leqslant j \leqslant n$, $1 \leqslant \mu \leqslant m$, $1 \leqslant \mu' \leqslant 2(m-\mu)$, $1 \leqslant \kappa \leqslant k$, $1 \leqslant \kappa' \leqslant 2(k-\kappa)$, and $0 \leqslant N \leqslant 3kn$. Therefore, the time complexity of each step of the binary search is $\mathcal{O}\left(n^2m^2k^3\right)$.

4. Analysis of a Special Case

We consider, in this section, a version of the problem $(Pm, Pk) \mid\mid C_{max}$ where all the tasks are accelerated when assigned to GPU, since this is the case in most applications, i.e $\underline{p_j} \leqslant \overline{p_j}$ for $j = 1, \ldots, n$ (all the tasks do not have the same acceleration factor on GPU). When considering this special case in the algorithm presented in the previous section, we note no amelioration in the time complexity of the algorithm, whereas the problem was simpler. Therefore we present an new algorithm for this case, based on a similar scheme, but with a much lower time complexity for an approximation ratio of $\frac{3}{2}$.

The algorithm is also based on the dual approximation technique. At each step, we have a guess on the optimal makespan. Let us consider one step of the dual approximation scheme and let as before λ denote the current guess.

The idea is to divide the set of tasks $\mathcal{T}$ into four sets of tasks, two of them whose tasks will be assigned to a CPU, $\mathcal{C}_1$, $\mathcal{C}_2$, and the other two whose tasks will be assigned to a GPU, $\mathcal{G}_1$ and $\mathcal{G}_2$. We denote the cardinality of set $\mathcal{C}_i$ (resp. $\mathcal{G}_i$) by $|\mathcal{C}_i|$ (resp. $|\mathcal{G}_i|$), $i = 1, 2$.

The first step of the algorithm consists in a preliminary assignment of the tasks of the whole set $\mathcal{T}$ to two of the sets: $\mathcal{G}_1$ and $\mathcal{C}_2$. This assignment of each task T_j is done by considering the value of the processing time $\overline{p_j}$ of T_j on CPU. The possible cases are described below:

- If $\overline{p_j} \leqslant \frac{\lambda}{2}$, task T_j is assigned to $\mathcal{C}_2$.
- Otherwise, $\frac{\lambda}{2} < \overline{p_j}$, task T_j is assigned to $\mathcal{G}_1$.

This assignment does not guarantee that the resulting schedule will have a makespan lower than $\frac{3}{2}\lambda$, even if there exists a schedule of makespan lower than λ. However, if $|\mathcal{G}_1| > k + m$, there are too many tasks with a processing time on CPU greater than $\frac{\lambda}{2}$ to fit in a schedule of makespan lower than λ. The dual approximation rejects this guess λ.

In the second step of the algorithm, in order to achieve the desired performance ratio, we have to reassign some of the tasks allocated to $\mathcal{C}_2$ to $\mathcal{G}_2$ and some of the tasks allocated to $\mathcal{G}_1$ to $\mathcal{C}_1$. We order the tasks of sets $\mathcal{C}_2$ and $\mathcal{G}_1$ in decreasing order of the computational surface change induced when a task T_j changes from a CPU to a GPU, $\overline{p_j} - \underline{p_j}$. One exception has to be made for set $\mathcal{G}_1$: some tasks can have a processing time on CPU larger than λ. These tasks are too big to fit on the CPUs with the current guess. They cannot be reassigned and are put at the end of $\mathcal{G}_2$, no matter the impact they can have on the computational surfaces. The tasks of the two sets will be examined in this new order. We can note that at most m tasks of set $\mathcal{G}_1$ can be reassigned to $\mathcal{C}_1$. The following steps are therefore repeated at most $m + 1$ times, as long as we have $W_C > m\lambda + \frac{\lambda}{2}$ or $W_G > k\lambda + \frac{\lambda}{2}$:

- While $W_G = \sum_{T_i \in \mathcal{G}_1 \cup \mathcal{G}_2} \underline{p_i} \leqslant k\lambda$, assign the first task of set $\mathcal{C}_2$ to $\mathcal{G}_2$.
- If $|\mathcal{C}_1| < m$, assign the first task from $\mathcal{G}_1$ to $\mathcal{C}_1$.

With this assignment, the computational area on the CPUs has been reduced to a minimum with the constraint of keeping the computational area on the GPUs lower than $k\lambda + \frac{\lambda}{2}$. Therefore, the value of W_C obtained by our algorithm is smaller than the value of the computational area on the CPUs of the optimal schedule, the most accelerated tasks having been assigned to the GPUs. Therefore, if $W_C > m\lambda + \frac{\lambda}{2}$

, we conclude that the value of λ is too small and adjust the bounds of our binary search accordingly.

If $W_C \leqslant m\lambda + \frac{\lambda}{2}$, we can construct a feasible schedule with a makespan lower than $\frac{3}{2}\lambda$ with the previous algorithm. Indeed, the number of tasks in $\mathcal{C}_1$ is lower than m so we can build a shelf S_1 as we did in Section 3, occupying $|\mathcal{C}_1|$ CPUs, with a length at most λ. The same arguments given in the proof of Lemma 1 can be used for building a shelf S_2 of length $\frac{\lambda}{2}$ and all the tasks from $\mathcal{C}_2$ can be fitted in the schedule as before. For the GPUs, the algorithm makes sure that the number of tasks in $\mathcal{G}_1$ is lower than k and that W_G does not go over the bound of $k\lambda + \frac{\lambda}{2}$ so shelves similar to S_1 and S_2 can be built easily. However, we did not have to make any discretization on the processing times of the tasks assigned to the GPUs here, so, contrary to Section 3, we get the same performance ratio of $\frac{3}{2}$ for any number k of GPUs. The time complexity of an algorithm based on this principle is in $\mathcal{O}(mn \log n)$.

5. Experiments

We run some experiments in order to show the efficiency of the proposed algorithmic scheme. They are two-fold, we first run a relaxed version of our algorithm by simulations on random instances and compared them to the classical reference algorithm HEFT [14] used on several actual systems (HEFT stands for Heterogeneous Earliest Finishing Time). Then, we implement the algorithm on the top of the xKaapi runtime system.

5.1. Preliminary: Analysis of HEFT.

HEFT proceeds in two phases, starting by a prioritization of the tasks that are sorted by decreasing average execution time and then the processor selection is obtained with the heterogeneous earliest finish time rule.

Lemma 3. *The worst case performance ratio of HEFT is larger than $m/2$.*

Proof. We show on the following instance that the prioritizing phase can provide a schedule whose makespan is far from the optimum. Let us consider an instance with a list of the following tasks: m tasks such that $\overline{p} = 1$ and $\underline{p} = \epsilon$ and for $i = 0, \cdots, m-1$: a single task of type $\mathcal{A}$ such that $\overline{p} = 1 - i/m$ and $\underline{p} = 1 - i/m$ and $m-1$ tasks of type $\mathcal{B}$ such that $\overline{p} = 1 - i/m$ and $\underline{p} = 1/m^2$. All these tasks are executed faster on the GPUs.

For this instance, HEFT fills first the m CPUs. Then, it fills alternatively the GPU with one task of type $\mathcal{A}$ and the m CPUs with m tasks of size $\mathcal{B}$. HEFT ends with a makespan equal to $m/2 + 3/2 - 1/m$. It is easy to check that the optimal makespan is equal to 1. □

5.2. Simulations.

The dual approximation based algorithm for problem $(Pm, Pk) \,||\, C_{max}$ presented in Section 3 provides a performance ratio of $\frac{4}{3} + \frac{1}{3k}$ with a reasonable time complexity. However, this running cost is not comparable to the one of HEFT which basically only needs to sort the tasks $(n \log n)$. This method can be modified in order to obtain a slightly worse performance ratio of 2 in a much slower time $\mathcal{O}(n^2 k)$, which would make it comparable to HEFT in terms of running time and still provide a performance guarantee. The idea is based on leaving aside the constraints ordering the tasks into shelves. The only constraint that remains is the one on the computational area on the GPUs being lower than $k\lambda$, λ being the current guess of the dual approximation. With the optimal computational area on the CPUs under this constraint determined by dynamic programming, we build a schedule with a makespan lower than twice the optimal value. This 2-approximation algorithm, denoted in short by DP in what follows, was implemented and compared

to HEFT by simulations based on various classes of instances. Moreover, for the special case where all the tasks are accelerated, we implemented the algorithm presented in Section 4, denoted by *Accel*, which provides a performance ratio of $\frac{3}{2}$ with a time complexity of $\mathcal{O}(mn \log n)$. All these algorithms are implemented in C++ programming language and run on a 3.4 GHz PC with 15.7 Gb RAM.

We report below a series of experiments run on random instances of various sizes: from 10 to 1000 tasks, with a step of 10 tasks, 2^a CPUs, a varying from 0 to 6, and 2^b GPUs, b varying from 0 to 3. For each combination of these sizes, 30 instances were considered, bringing us to a total of 10500 tested instances. The processing times on the CPUs are randomly generated using the uniform distribution $U[10, 100]$. The distribution of the acceleration factors on the GPUs has been measured in [24] using the classical numerical kernels of Magma [25] in a multi-core multi-GPU machine hosted by the Grid'5000 infrastructure experimental platform [26]. We extracted a distribution of the acceleration factors which reflects the qualitative speed-up on classical numerical kernels: we assign to each task an acceleration factor $\alpha_j = \frac{p_j}{\overline{p_j}}$ of 1/15 or 1/35 with a probability of 1/2. The resulting processing times on the GPUs are thus $\underline{p_j} = \alpha_j \overline{p_j}$. Since in this generation scheme all the tasks of these instances are accelerated on GPU, DP, HEFT and Accel were all compared on these instances. The running time of the three algorithms is always under one second, even for the largest instances. We calculated the mean and maximal deviations of the makespans of the solutions returned by these algorithms from the lower bound of the makespan derived from the binary search of the approximation algorithm, over all the instances.

TABLE 1. Maximal deviations (%) for DP and HEFT

n	120	160	220	260	360	380	660	700	760	780	920	940
DP	76.88	72.73	70.37	69.14	70.00	70.00	67.42	50.82	42.77	54.47	91.77	63.07
HEFT	123.53	98.44	92.55	91.90	110.37	91.78	113.48	98.10	98.77	103.15	116.46	96.31
Accel	46.15	42.86	50.00	41.18	37.82	43.59	36.36	40.91	32.52	37.04	48.24	34.65

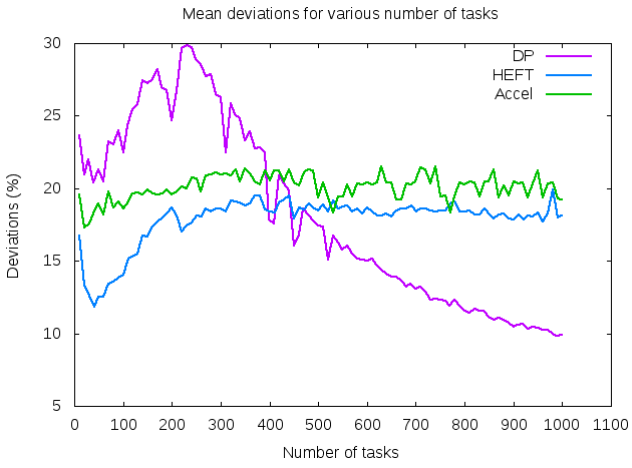

FIGURE 3. Mean deviations of DP and HEFT for various n

As can seen in Table 1, the maximal deviations of DP are usually below the maximal deviations of HEFT and more importantly, these deviations respect the theoretical performance guarantee in the case of DP whereas the maximal deviations of HEFT sometimes go over the 100% barrier corresponding to a performance ratio

of 2. The same can be said for Accel, with maximal deviations staying below the 50% barrier corresponding to a performance ratio of $\frac{3}{2}$.

Figure 3 shows that in average, DP even outperforms HEFT for large instances. However, Accel remains slightly above HEFT, staying close to its $\frac{3}{2}$ bound but never going above it, contrary to HEFT which does not have a performance guarantee, as seen in Table 1. This better performance ratio of $\frac{3}{2}$ with its low cost make Accel preferable to DP for practical intensive use with still a better maximum performance guarantee.

5.3. **Experiments on a real run-time.** We investigate in this section the practical use of the algorithms of the previous family. We target classical linear algebra kernels, since they are extensively used and they generate a loop of independent tasks. Here, the computations consist in a series of independent tasks. DP (with ratio 2), HEFT and $\frac{4}{3}$-approximation algorithms were implemented in the scheduler of the xKaapi run-time system [11].

Implementation of the $\frac{4}{3}$-approximation algorithm. In most linear algebra applications, the block size is the most important characteristic to maximize the performance. Indeed, the block size has a direct impact on memory transfers between the host and the accelerators, cache effects and task graph size .

In the following experiments of Figure 4, we study the variation of the computation time as a function of the block size for the same matrix size of a Cholesky factorisation extracted from the MAGMA library. We use a single GPU and the matrix is decomposed over simple square blocks. The experiments have been conducted on a quad-core Intel i7-3840QM with hyperthreading and a Nvidia Quadro K1000M GPU.

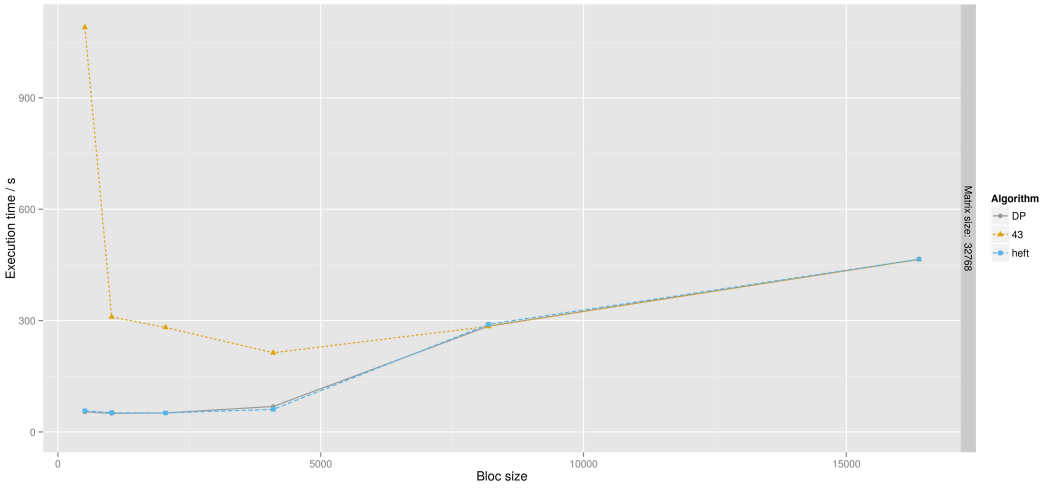

FIGURE 4. Execution time of a Cholesky factorization scheduled by DP, 4/3 and HEFT for various block sizes, on 3 hyperthreaded core and a single GPU

As the block size decreases, the number of independent tasks increases. Thus, the computation time of the scheduling using the $\frac{4}{3}$-approximation algorithm increases quadratically with the number of tasks too. As a result, the scheduling time dominates the execution time saved for large block size (which corresponds to small number of tasks). The $\frac{4}{3}$-approximation algorithm is therefore usable mostly for

cases where the computation time is larger than the scheduling time. It is probably not the best suited algorithm for linear algebra kernels.

Practical issues: DP versus HEFT. We compare now the relaxed version of the algorithm (DP) with HEFT on a machine with several GPUs. The experiments have been conducted on a heterogeneous, multi-GPU system composed of two six-core Intel Xeon X5650 CPUs running at 2.66 GHz with 72 GB of memory. This parallel system is enhanced with eight NVIDIA Tesla C2050 GPUs (Fermi architecture) of 448 GPU cores (scalar processors) running at 1.15 GHz each (2688 GPU cores total) with 3 GB GDDR5 per GPU (18 GB in total). It has 4 PCIe switches to support up to 8 GPUs. When 2 GPUs share a switch, their aggregated PCIe bandwidth is bounded by the one of a single PCIe 16x.

Algorithm	Gflops	Memory transfer/GB
HEFT	535	2.62
DP	565	1.91

TABLE 2. Perfomance of DP and HEFT for Cholesky factorization with m=4 CPUs and k=8 GPUs

The implementation of DP was combined with an improved local mapping for minimizing the data transfers [27]. With 8 GPUs, DP outperforms HEFT both in the raw performance and memory transfers (see Table 2). The execution times are close to each other in all cases, but DP has the three major following advantages:

- the guarantee on the scheduling results,
- a decrease in the volume of communication,
- no need of accurate communication model.

6. Concluding Remarks

In this paper, we presented a new scheduling algorithm using a generic methodology (in the opposite of specific *ad hoc* algorithms) for hybrid architectures (multicore machine with GPUs). We proposed fast algorithms with a constant approximation ratio in the case of independent tasks and an faster version in the special case where all the tasks are accelerated when assigned to GPU. A ratio of $\frac{4}{3} + \frac{1}{3k} + \epsilon$ is achieved for k GPUs in the general case, and a ratio of $\frac{3}{2} + \epsilon$ when the tasks are all accelerated on GPU. The main idea of the approach is to determine an adequate partition of the set of tasks on the CPUs and the GPUs using a dual approximation scheme. A simulation and experimental analysis on a real run-time systemm have been provided to assess the computational efficiency of the proposed methods. The main conclusion is that these algorithms are stable because of their approximation guaranties, however, the high running time is often dominated by the cost of the scheduling it-self, leading to inefficiency if the number of tasks is too small. According to our experimental setting, the relaxed version with approximation ratio equal to 2 was the best trade-off.

As further investigations of this work, we plan to extend the analysis to more generic problems where the tasks are dependent with larger number of tasks. We also plan on testing the robustness of the presented algorithms to perturbations in the execution times that are only estimated on real-life computing platforms.

References

[1] Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupaty S, Hammarlund P, *et al.*. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ISCA*, Seznec A, Weiser UC, Ronen R (eds.), ACM, 2010; 451–460.

[2] Hochbaum DS, Shmoys DB. A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing* 1988; **17**(3):539–551.
[3] Bonifaci V, Wiese A. Scheduling unrelated machines of few different types. *CoRR* 2012; **abs/1205.0974**.
[4] Pinel F, Dorronsoro B, Bouvry P. Solving very large instances of the scheduling of independent tasks problem on the gpu. *Journal of Parallel Distrib. Comput.* 2012; .
[5] Agullo E, Augonnet C, Dongarra J, Faverge M, Ltaief H, Thibault S, Tomov S. Qr factorization on a multicore node enhanced with multiple gpu accelerators. *IEEE Int. Parallel & Distributed Processing Symposium (IPDPS)*, 2011.
[6] Song F, Tomov S, Dongarra J. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. *26th ACM International Conference on Supercomputing (ICS 2012)*, ACM: Venice, Italy, 2012.
[7] Boukerche A, Correa JM, Melo A, Jacobi RP. A hardware accelerator for the fast retrieval of dialign biological sequence alignments in linear space. *IEEE Transactions on Computers* 2010; **59**:808–821.
[8] Phillips JC, Stone JE, Schulten K. Adapting a message-driven parallel application to gpu-accelerated clusters. *SC*, 2008.
[9] Bueno J, Planas J, Duran A, Badia RM, Martorell X, Ayguadé E, Labarta J. Productive programming of gpu clusters with ompss. *IPDPS*, IEEE Computer Society, 2012; 557–568.
[10] Augonnet C, Thibault S, Namyst R, Wacrenier PA. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 2011; **23**:187–198.
[11] Gautier T, Ferreira L, Joao V, Maillard N, Raffin B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. *Proc. of IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, Boston, USA, 2013.
[12] Chen L, Ye D, Zhang G. Online scheduling on a cpu-gpu cluster. *TAMC* 2013; **7876**:1–9.
[13] Hochbaum DS, Shmoys DB. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM* 1987; **34**(1):144–162.
[14] Topcuoglu H, Hariri S, Wu MY. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE TPDS* 2002; **13**(3):260–274.
[15] Kedad-Sidhoum S, Monna F, Mounié G, Trystram D. Scheduling independent tasks on multi-cores with gpu accelerators. *Proc. HeteroPar 2013, Aachen* August 2013; :228–237.
[16] Garey MR, Grahams RL. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing* 1975; **4**:187–200.
[17] Blazewicz J, Ecker K, Pesch E, Schmidt G, Weglarz J. *Handbook on Scheduling, From Theory to Applications, International Handbooks on Information Systems*. Springer, 2007.
[18] Lenstra JK, Shmoys DB, Tardos E. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming* 1988; **46**:259–271.
[19] Shmoys DB, Tardos E. An approximation algorithm for the generalized assignment problem. *Mathematical Programming* 1993; **62**:461–474.
[20] Shchepin EV, Vakhania N. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters* 2004; **33**:127–133.
[21] Friesen DK. Tighter bounds for lpt scheduling on uniform processors. *SIAM Journal on Computing* 1987; **16**(3):554–560.
[22] Nélis V, Raravi G. A ptas for assigning sporadic tasks on two-type heterogeneous multiprocessors. *RTSS* 2012; .
[23] Imreh C. Scheduling problems on two sets of identical machines. *Computing* 2003; **70**:277–294.
[24] Seifu S. Scheduling on heterogeneous cluster environments. Master's Thesis, Grenoble university Jun 2012.
[25] Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series* 2009; **180**.
[26] Bolze R, Cappello F, Caron E, Daydé MJ, Desprez F, Jeannot E, Jégou Y, Lanteri S, Leduc J, Melab N, *et al.*. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *IJHPCA* 2006; **20**(4):481–494.
[27] Bleuse R, Gautier T, Lima JF, Mounié G, Trystram D. Scheduling data flow program in xkaapi: A new affinity-based algorithm for heterogeneous architectures. *20th International European Conference on Parallel Processing*, ARCoSS/LNCS, Springer: Porto, Portugal, 2014. To appear.