Scheduling Independent Tasks on Multi-cores with GPU Accelerators

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

Scheduling Independent Tasks on Multi-Cores with GPU Accelerators

Safia Kedad-Sidhoum[1], Florence Monna[1]*, Grégory Mounié[2], and Denis Trystram[2],[3]

[1] Laboratoire d'Informatique de Paris 6, 4 Place Jussieu, 75005 Paris.
[2] Univ. Grenoble-Alpes, 655 avenue de l'Europe, 38334 St. Ismier, France.
[3] Institut Universitaire de France.

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 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 a dynamic programming scheme to balance evenly the load between the heterogeneous resources. Finally, we run some simulations based on realistic benchmarks and compare the solution obtained by a relaxed version of this method to the one provided by a classical greedy algorithm and to lower bounds on the value of the optimal makespan.

1 Introduction

Most of the computing systems available today include parallel multi-core chips sharing a large memory with additional hardware accelerators. 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 these new features, effective and automatic management of the hybrid resources will be more and more important for running applications. These new architectures have given rise to new scheduling problems. The main challenge is to create adequate generic methods and software tools that fulfill the requirements for optimizing the performances.

There exist in the literature a huge number of papers dealing with implementations of specific applications using GPUs or hybrid CPU-GPU architectures. They consist mostly in studying the gains and performances of the parallelisation of some specific numerical kernels [1] or specific applications like multiple alignments of biological sequences [2], molecular dynamics, etc. Moreover, most

* This work is supported by a CNRS-Google contract and a french national program GDR-RO.

of existing scheduling algorithms and tools are not well-suited for general purpose applications since the architecture of the GPUs differs from CPUs and thus, the GPUs should be considered as a new type of resources to develop efficient approaches. New features have been implemented in some parallel programming environments and runtime systems: they provide hybrid (CPU/GPU) programming operators, automatic scheduling and adequate data movements. For instance, OmpSs [3], StarPU [4] or xKaapi [5] include scheduling policies that are restricted to fast greedy algorithms or work stealing.

Our objective within this work is to propose a new algorithm for a general purpose scheduling for the execution of independent tasks on hybrid CPU-GPU architectures designed for High Performance Computing (HPC). The considered input is a set of independent sequential tasks whose execution times are known. This hypothesis is realistic, since some computing platforms such as StarPU have a module which estimates at compile time the different execution times of the considered tasks. The method that we propose in this work determines the allocation and schedule of the tasks to the computing units, CPUs and GPUs. We analyze in detail this methodology for the case of 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 [6] with a dynamic programming scheme. The cost of the algorithm is in $\mathcal{O}\left(n^2k^3m^2\right)$. As this method is costly, we derive a relaxed algorithm and compare it experimentally with one of the most popular algorithm (HEFT [7]).

The outline of the paper is as follows. In Section 2, a formal description of the scheduling problem with k GPUs is provided and some related works are presented. We propose in Section 3 a new approach for solving the problem and its analysis in Section 4. We report the results of experiments in Section 5 where a relaxed version of our method is compared to the classical HEFT algorithm on simulations built from realistic workloads. The experimental analysis shows that the proposed method has a more stable behavior than HEFT for a similar performance. Finally, some perspectives are discussed in Section 6.

2 Problem Definition and Related Works

We consider a multi-core parallel platform with 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 sequential tasks denoted by $T_1, \ldots, T_n$. 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. We assume that both processing times of a task are known in advance (or at least can be estimated at compile time). The makespan is defined as the maximum completion time of the last finishing task. For the problem considered, in the optic of High Performance Computing, the objective is to minimize the makespan of the schedule. The problem will be denoted by $(Pm, Pk) \mid\mid C_{max}$.

Observe that if both processing times are equal ($\overline{p_j} = \underline{p_j}$) for $j = 1, \ldots, n$, $(Pm, P1)$ $||$ C_{max} is equivalent to the classical P $||$ C_{max} problem, which is NP-hard. Thus, the problem of scheduling with GPUs is also NP-hard and we are looking for efficient approximation algorithms.

(Pm, Pk) $||$ C_{max} is a special case of R $||$ C_{max}. Lenstra et al. [8] propose a PTAS for the problem R $||$ $C_{\max}$ with running time bounded by the product of $(n+1)^{m/\epsilon}$ and a polynomial of the input size. Let us notice that if parameter m is not fixed, then the algorithm is not fully polynomial. The authors also prove that unless $P = NP$, there is no polynomial-time approximation algorithm for R $||$ $C_{\max}$ with an approximation factor less than $3/2$ and present a 2-approximation algorithm. This algorithm is based on rounding the optimal solution of the preemptive version of the problem. Shmoys and Tardos [9] generalize this technique to obtain the same approximation factor for the generalized assignment problem. Furthermore, they generalize the rounding technique to hold for any fractional solution. Recently, Shchepin and Vakhania [10] introduce a new rounding technique which yields an improved approximation factor of $2 - \frac{1}{m}$. This is so far the best approximation result for R $||$ $C_{\max}$. If we look at the more specific problem of scheduling unrelated machines of few different types, Bonifaci and Wiese [11] present a PTAS to solve this problem. However, the time complexity of the polynomial algorithm is not provided so that the algorithm does not seem to be potentially useful from a practical perspective. Finally, it is worth noticing that if all the tasks of the addressed problem have the same acceleration on the GPUs, the problem reduces to a Q $||$ $C_{\max}$ problem, with two machines speeds. The first PTAS for Q $||$ $C_{\max}$ was given by Hochbaum and Shmoys [12]. The overall running time of the algorithm is $\mathcal{O}((\log m + \log(3/\epsilon))(m/\epsilon)(n/\epsilon)1/\epsilon)$.

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 Rationale of the solving method

The principle of the algorithm is to use the dual approximation technique [6]. A g-dual approximation algorithm for a generic problem takes a real number λ (guess) as an input and either delivers a schedule of makespan at most $g\lambda$, or answers correctly that there exists no schedule of length at most λ.

We target $g = \frac{4}{3} + \frac{1}{3k}$. Let λ be the current real number input for the dual approximation. In the following, we assert that there exists a schedule of length lower than λ. Then, we have to show how it is possible to build a schedule of length at most $\frac{4\lambda}{3} + \frac{1}{3k}$.

The idea of the algorithm 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. 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. The way to determine the partition is described in Section 4.2.

4 Theoretical Analysis

4.1 Structure of an Optimal Schedule

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 being 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: $W_C = \sum_{j \,/\, \pi(j) \in \mathcal{C}} \overline{p_j}$.

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 is at most $m\lambda$, and the computational area on the GPUs is at most $k\lambda$.

Property 2. In an optimal solution, if there exist two consecutive tasks on a CPU, if one of these tasks has an execution time greater than $\frac{2\lambda}{3}$, then the other one has an execution time lower than $\frac{\lambda}{3}$. The same can be said of two consecutive tasks on a GPU.

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

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 (respectively on GPU) are strictly greater than $\frac{2\lambda}{3}$ do not use more than m CPUs (respectively k GPUs), and hence can be executed concurrently in the first set in a shelf denoted by S_1 (respectively S_5). We denote by μ the number of CPUs and κ the number of GPUs executing these tasks.

The tasks whose execution times are lower than $\frac{2\lambda}{3}$ and strictly greater than $\frac{\lambda}{3}$ on CPU (respectively on GPU) cannot be executed on the μ CPUs occupied by S_1 from Property 1 (respectively the κ GPUs occupied by S_5). Moreover, from Property 3, $2(m-\mu)$ of these tasks on CPU (respectively $2(k-\kappa)$ tasks on GPU) can be executed in time at most $\frac{4\lambda}{3}$ on the remaining $(m-\mu)$ CPUs in

the second set and fill two shelves S_3 and S_4 of equal length $\frac{2\lambda}{3}$ (resp. on $(k-\kappa)$ GPUs and fill two similar shelves S_7 and S_8).

The remaining tasks have execution times lower than $\frac{\lambda}{3}$ on CPU (resp. on GPU) and can be executed within a time at most $\frac{\lambda}{3}$ in the first set on the CPUs in another shelf denoted by S_2 (resp. on the GPUs in a shelf S_6).

Thus, we are looking for a schedule on the CPUs in two sets of two shelves: S_1 of length λ, S_2 as well as S_3 and S_4 of length $\frac{2\lambda}{3}$, and a similar schedule on the GPUs, with 4 shelves S_5 to S_8.

Lemma 1. *The length of S_2 is lower than $\frac{\lambda}{3}$.*

Proof. We start by modifying the starting times of the tasks in S_2, in order to have all the tasks justified to the right of the schedule, so that all the processors complete their tasks exactly at time $\frac{4\lambda}{3}$ (like in figure 1). This operation induces on each CPU an idle time interval between the completion of the tasks of S_1 and the starting of the tasks of S_2. We define the load of a CPU as the sum of the execution times of the tasks processed on it. By definition the load is equal to $\frac{4\lambda}{3}$ minus the length of the idle time interval on the CPU. Now consider the following algorithm to schedule the tasks of processing time lower than $\frac{\lambda}{3}$:

- Consider the tasks in an arbitrary order $T_1, \ldots, T_f$, f being the total number of tasks remaining to be allocated in S_2.
- Allocate task T_i to the least loaded processor, at the latest possible date. Update its load.

The only problem that may occur when allocating task T_i is that T_i cannot be completed before the starting time of the tasks of S_2. However at each step, the least loaded processor has a load at most λ; otherwise according to Property 1 it would contradict the fact that the total work area of the tasks is bounded by $m\lambda$. Hence, the idle time interval on the least loaded CPU has a length at least $\frac{\lambda}{3}$ and can contain T_i. □

The proof is similar for the shelf S_6 and will be detailed in section 4.2.

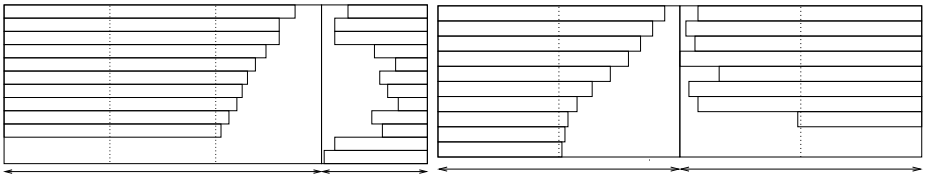
4.2 Partitioning the Tasks into Shelves

In this section, we detail how to fill the shelves (see Figure 1) on the CPUs and on the 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 and the same sets on the GPUs, we look for an assignment satisfying the following four 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 uses a total of at most m processors. We still denote by μ the number of processors they use. These tasks are intended to be scheduled in S_1.

- (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 uses a total of at most $2(m-\mu)$ processors. These tasks are intended to be scheduled in S_3 and S_4.
- (C_4) The total computational area on the GPU 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 uses a total of at most k processors. We still denote by κ the number of processors they use. These tasks are intended to be scheduled in S_5.
- (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 uses a total of at most $2(k-\kappa)$ processors. These tasks are intended to be scheduled in S_7 and S_8.



(a) Shelves S_1 and S_2 (b) Shelves S_3 and S_4

Fig. 1. Partitioning the set of tasks into shelves

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 the GPU. Determining if an allotment satisfying (C_1), (C_3), (C_4) and (C_6) exists reduces to solving a three-dimensional knapsack problem that can be formulated as follows:

$$
\begin{aligned}
W_C^* = & \min \sum_{j=1}^{n} \overline{p_j} x_j && (C_1)\\
\text{s.t. } & \frac{1}{2} \sum_{2\lambda/3 \geqslant \overline{p_j} > \lambda/3} x_j + \sum_{\overline{p_j} > 2\lambda/3} x_j \leqslant m && (C_3)\\
& \frac{1}{2} \sum_{2\lambda/3 \geqslant \underline{p_j} > \lambda/3} (1 - x_j) + \sum_{\underline{p_j} > 2\lambda/3} (1 - x_j) \leqslant k && (C_6)\\
& \sum_{j=1}^{n} \underline{p_j} \, (1 - x_j) \leqslant k\lambda && (C_4)\\
& x_j \in \{0, 1\}
\end{aligned}
$$

We propose a dynamic programming algorithm that solves the knapsack problem in $\mathcal{O}\left(n^2 m^2 k^3\right)$. For this purpose, we first have to reduce the states on the GPUs

to a smaller number. We use the time intervals of length $\frac{\lambda}{3n}$ and introduce the integer number ν_j of these time intervals required for a task T_j if it is executed on the GPUs: $\nu_j = \left\lfloor \frac{p_j}{\lambda/(3n)} \right\rfloor$. $N = \sum_{\pi(j)\in\mathcal{G}} \nu_j$ denotes the total integer number of these intervals on the GPUs. We can define the error on the processing time of each task $\epsilon_j = p_j - \nu_j \frac{\lambda}{3n}$ created by this approximation.

This result allows us to consider only N states in the dynamic programming regarding the workload on the GPUs, and 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 $n\frac{\lambda}{3n} = \frac{\lambda}{3}$. Then, constraint (C_4) of the linear program becomes

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

The truncated computational area of the GPUs is at most $k\lambda$ and thus, the full computational area remains lower than $k\lambda + \frac{\lambda}{3}$. Thus, an upper bound on the length of shelf S_6 can be determined as follows.

Lemma 2. *The length of S_6 is lower than $\frac{\lambda}{3} + \frac{\lambda}{3k}$.*

Proof. The proof is similar to the one of Lemma 1. 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 S_2, the only problem that may occur is again that 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 $\lambda + \frac{\lambda}{3k}$; otherwise it would contradict the fact that 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. So S_6 has a length lower than $\frac{\lambda}{3} + \frac{\lambda}{3k}$. □

Once this reduction done, 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 assigned to the CPUs, μ of them having processing times $\overline{p_j}$ greater than $\frac{2\lambda}{3}$ and μ' with $\frac{\lambda}{3} < \overline{p_j} \leqslant \frac{2\lambda}{3}$, respectively κ and κ' of these tasks being assigned to the GPUs such that N time intervals are occupied on the GPUs.

We use a dynamic programming which allows us 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. If task T_j is assigned to a CPU, the resulting sum of all the processing times of the tasks on the CPUs is then

$$F_{CPU}(j, \mu, \mu', \kappa, \kappa', N) = \overline{p_j} + W_C\left(j-1, \mu - I_{\left(\overline{p_j} > \frac{2\lambda}{3}\right)}, \mu' - I_{\left(\frac{2\lambda}{3} \geqslant \overline{p_j} > \frac{\lambda}{3}\right)}, \kappa, \kappa', N\right)$$

where $I_{(\overline{p_j}>\frac{2\lambda}{3})}$ and $I_{(\frac{2\lambda}{3}\geqslant\overline{p_j}>\frac{\lambda}{3})}$ are indicating functions:

$$I_{(\overline{p_j}>\frac{2\lambda}{3})}=\begin{cases}1 & \text{if } \overline{p_j}>\frac{2\lambda}{3}\\ 0 & \text{otherwise}\end{cases}, \quad I_{(\frac{2\lambda}{3}\geqslant\overline{p_j}>\frac{\lambda}{3})}=\begin{cases}1 & \text{if } \frac{2\lambda}{3}\geqslant\overline{p_j}>\frac{\lambda}{3}\\ 0 & \text{otherwise}\end{cases}$$

If task T_j is assigned to a GPU, the sum of all the processing times of the tasks on the CPUs is then

$$F_{GPU}(j,\mu,\mu',\kappa,\kappa',N)=W_C\left(j-1,\mu,\mu',\kappa-I_{(\underline{p_j}>\frac{2\lambda}{3})},\kappa'-I_{(\frac{2\lambda}{3}\geqslant\underline{p_j}>\frac{\lambda}{3})},N-\nu_j\right),$$

with $I_{(\underline{p_j}>\frac{2\lambda}{3})}=\begin{cases}1 & \text{if } \underline{p_j}>\frac{2\lambda}{3}\\ 0 & \text{otherwise}\end{cases}$, $I_{(\frac{2\lambda}{3}\geqslant\underline{p_j}>\frac{\lambda}{3})}=\begin{cases}1 & \text{if } \frac{2\lambda}{3}\geqslant\underline{p_j}>\frac{\lambda}{3}\\ 0 & \text{otherwise}\end{cases}$.

The dynamic programming is then based on the following recursive equation: for $1\leqslant j\leqslant n$, $1\leqslant\mu\leqslant m$, $1\leqslant\mu'\leqslant 2(m-\mu)$, $1\leqslant\kappa\leqslant k$, $1\leqslant\kappa'\leqslant 2(k-\kappa)$, $0\leqslant N\leqslant 3kn$,

$$W_C(j,\mu,\mu',\kappa,\kappa',N)=\min\left(F_{CPU}(j,\mu,\mu',\kappa,\kappa',N),F_{GPU}(j-1,\mu,\mu',\kappa,\kappa',N-\nu_j)\right)$$

In order to satisfy the constraints imposing that $\mu\leqslant m$ tasks with a processing time greater than $\frac{2\lambda}{3}$ are processed on the CPUs and no more than $2(m-\mu)$ tasks with a processing time lower than $\frac{2\lambda}{3}$ and greater than $\frac{\lambda}{3}$ are processed on the CPUs and that the computational area of the GPUs is not greater than $\frac{4k\lambda}{3}$, we have border conditions:

$$W_C(j,\mu,\mu',\kappa,\kappa',N)=+\infty \begin{cases}\text{if } \mu>m \text{ or } \mu'>2(m-\mu)\\ \text{if } \kappa>k \text{ or } \kappa'>2(k-\kappa)\\ \text{if } \sum\limits_{\pi(j)\in\mathcal{G}}\nu_j>3kn\end{cases}$$

If the optimal value of the computational area on the CPUs $W_C^*=W_C(n,\mu,\mu',\kappa,\kappa',N)$, for $1\leqslant\mu\leqslant m$, $1\leqslant\mu'\leqslant 2(m-\mu)$, $1\leqslant\kappa\leqslant k$, $1\leqslant\kappa'\leqslant 2(k-\kappa)$, $0\leqslant N\leqslant 3kn$, is greater than $m\lambda$, then there exists no solution with a makespan at most λ, and the algorithm answers "NO" to the dual approximation. Otherwise, we construct a feasible solution with a makespan at most $\frac{4\lambda}{3}+\frac{\lambda}{3k}$, with the corresponding shelves on the CPUs and the corresponding μ, μ',κ, κ' and N values.

We have described one step of the dual-approximation algorithm, with a fixed guess. A binary search will be used to try different guesses to approach the optimal makespan as follows.

Binary Search. We first take an initial lower bound B_{min} and an initial upper bound B_{max} of our optimal makespan. We start by solving the problem with λ equal to the average of these two bounds and then we adjust the bounds: if the previous algorithm returns "NO", then λ becomes the new lower bound, but if the algorithm returns a schedule of makespan at most $\frac{4\lambda}{3}$, then λ becomes

the new upper bound. The number of iterations of this binary search can be bounded by $\log(B_{max} - B_{min})$.

Cost Analysis. Solving the dynamic program 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 $\mathcal{O}\left(n^2m^2k^3\right)$.

5 Experimental Analysis

In order to show the efficiency of our method, we run experiments on random instances and compared them to the classical reference greedy algorithm HEFT used on several actual systems (HEFT stands for Heterogeneous Earliest Finishing Time [7]). For instance, the scheduling decisions in StarPU [4] are based on estimations of the execution times of the tasks on both CPU and GPU resources that are scheduled by HEFT.

5.1 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. We can note that HEFT is not a list scheduling algorithm since some computing resources may stay idle even if a task could be executed on it.

If we consider the problem with m and $k = 1$, HEFT leads to a solution with a makespan that can be as far as a ratio of $m/2$ from the optimal value while the worst case performance ratio of our algorithm is bounded by a small constant.

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 $\underline{p} = \epsilon$.
- for $i = 0, \cdots, m-1$:
  - a single task of type $\mathcal{A}$ such that $\overline{p} = 1 - i/m$ and $\underline{p} = 1 - i/m$;
  - $m-1$ tasks of type $\mathcal{B}$ such that $\overline{p} = 1 - i/m$ and $\underline{p} = 1/m^2$. These tasks are executed faster on the GPUs.

On 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 $OPT = 1$. □

5.2 Computational Experiments

The HEFT algorithm is an efficient algorithm in practice but it does not provide any performance guarantee. The algorithm we presented in the previous section on the other hand provides a very satisfying performance ratio of $\frac{4}{3} + \frac{1}{3k}$, can be implemented and works in a reasonable amount of time, but its running time may make it a bit slow for users who want an quick response, and do not necessarily need a performance guarantee so precise of $\frac{4}{3} + \frac{1}{3k}$.

The method described before can be modified in order to obtain a performance ratio of 2 in a time $\mathcal{O}(n^2k)$, 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 can build a schedule with a makespan lower than twice the optimal value.

This algorithm was implemented and compared to HEFT with an experimental analysis based on various classes of instances. All the algorithms are implemented in C++ programming language and run on a 3.4 GHz PC with 15.7 Gb RAM.

We run a series of experiments on random instances of different sizes: 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 instances tested in total. The processing times on the CPUs are randomly generated using the uniform distribution U[10,100] so that $\overline{p_j} \in \{1, \ldots, 100\}$ for each task T_j. The distribution of the acceleration factors on the GPUs have been measured in [13] using the classical numerical kernels of Magma [14] in a multi-core multi-GPU machine hosted by the Grid'5000 infrastructure experimental platform [15]. We extracted a distribution of the acceleration factors which reflects the qualitative speed-up on real kernels: we assign to each task an acceleration factor α_j of 1/15 or 1/35 with a probability of 1/2. The resulting processing times on the GPUs are thus $\underline{p_j} = \alpha_j \overline{p_j}$. We calculated the mean, maximal and minimal deviations of the Dynamic Programming (DP) based approximation algorithm and the HEFT algorithm from the optimal values derived from the binary search of our approximation programming.

n	120	160	220	260	280	360	380	660	700	760	780	920	940
DP	76.88	72.73	70.37	70.00	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	94.34	91.90	110.37	91.78	113.48	98.10	98.77	103.15	116.46	96.31

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

As we can see 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

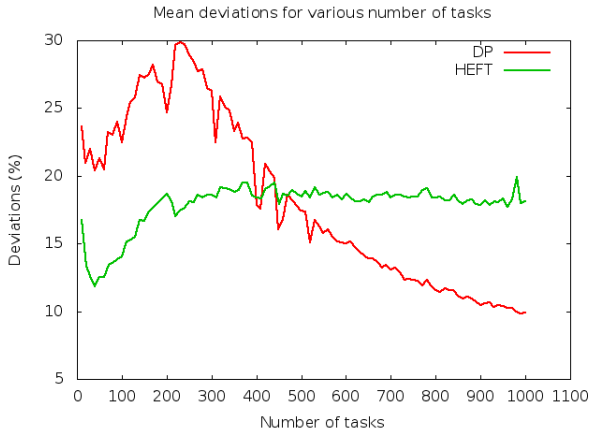


Fig. 2. Mean deviations of DP and HEFT for various n

deviations of HEFT sometimes go over the 100% barrier corresponding to a performance ratio of 2. We can see in Figure 2 that on average, DP even outperforms HEFT for large numbers of tasks.

6 Concluding Remarks

In this paper, we presented an analysis for scheduling algorithms using a generic methodology (in the opposite of specific *ad hoc* algorithms). We proposed fast algorithms with a constant approximation ratio in the case of independent tasks on a multi-core machines with GPUs. In the case of a single (resp. multiple) GPU(s) a ratio of $\frac{4}{3} + \epsilon$ (resp. $\frac{4}{3} + \frac{1}{3k} + \epsilon$) is achieved. 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 have been provided for different kernels to assess the computational efficiency of the proposed methods. We are currently implementing the algorithms presented here in the runtime systems xKaapi and StarPU.

As further investigations of this work, we plan to extend the analysis to more generic problems where the tasks are linked by a precedence relation. We believe that the same algorithmic scheme could be adapted to provide faster scheduling algorithms at a price of a worse guarantee. For instance, we expect a ratio 2 with a complexity of $(O)(nlog(n))$. These algorithms built using the proposed scheme are good candidates for the integration into runtime systems like StarPU and xKaapi.

References

1. Song, F., Tomov, S., Dongarra, J.: Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In: 26th ACM International

Conference on Supercomputing (ICS 2012), Venice, Italy, ACM (June 2012)
2. Boukerche, A., Correa, J.M., Melo, A., Jacobi, R.P.: A hardware accelerator for the fast retrieval of dialign biological sequence alignments in linear space. IEEE Transactions on Computers **59** (2010) 808–821
3. Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguadé, E., Labarta, J.: Productive programming of gpu clusters with ompss. In: IPDPS, IEEE Computer Society (2012) 557–568
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23** (2011) 187–198
5. Gautier, T., Ferreira, L., Joao, V., Maillard, N., Raffin, B.: Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: Proc. of IEEE Int. Parallel and Distributed Processing Symposium (IPDPS), Boston, USA (2013)
6. Hochbaum, D.S., Shmoys, D.B.: Using dual approximation algorithms for scheduling problems theoretical and practical results. J. ACM **34**(1) (1987) 144–162
7. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE TPDS **13**(3) (2002) 260–274
8. Lenstra, J.K., Shmoys, D.B., Tardos, E.: Approximation algorithms for scheduling unrelated parallel machines. Mathematical Programming **46** (1988) 259–271
9. Shmoys, D.B., Tardos, E.: An approximation algorithm for the generalized assignment problem. Mathematical Programming **62** (1993) 461–474
10. Shchepin, E.V., Vakhania, N.: An optimal rounding gives a better approximation for scheduling unrelated machines. Operations Research Letters **33** (2004) 127–133
11. Bonifaci, V., Wiese, A.: Scheduling unrelated machines of few different types. CoRR **abs/1205.0974** (2012)
12. Hochbaum, D.S., Shmoys, D.B.: A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. SIAM Journal on Computing **17**(3) (1988) 539–551
13. Seifu, S.: Scheduling on heterogeneous cluster environments. Master's thesis, Grenoble university (Jun 2012)
14. 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 **180** (2009)
15. Bolze, R., Cappello, F., Caron, E., Daydé, M.J., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quétier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. IJHPCA **20**(4) (2006) 481–494