Scheduling Divisible Loads on Heterogeneous Desktop Systems with Limited Memory

Aleksandar Ilic and Leonel Sousa

INESC-ID, IST/UTLisbon
Rua Alves Redol, 9, 1000-029 Lisbon, Portugal
{Aleksandar.Ilic,Leonel.Sousa}@inesc-id.pt

Abstract. This paper addresses the problem of scheduling discretely divisible applications in heterogeneous desktop systems with limited memory by relying on realistic performance models for computation and communication, through bidirectional asymmetric full-duplex buses. We propose an algorithm for multi-installment processing with multi-distributions that allows to efficiently overlap computation and communication at the device level in respect to the supported concurrency. The presented approach was experimentally evaluated for a real application; 2D FFT batch collaboratively executed on a Graphic Processing Unit and a multi-core CPU. The experimental results obtained show the ability of the proposed approach to outperform the optimal implementation for about 4 times, whereas it is not possible with the current state of the art approaches to determine a load balanced distribution.

Keywords: Scheduling, divisible loads, heterogeneous desktop systems, multiple installments, memory constraints.

1 Introduction

Modern desktop systems are already true heterogeneous platforms capable of sustaining remarkable computation power by coalescing the execution space of multi-core CPUs and programmable accelerators, such as Graphics Processing Units (GPUs) [9]. In this paper, we consider the problem of scheduling discretely divisible load (DL) applications on heterogeneous desktop platforms, from the perspective of employing all the available computing devices. The DL model [12] represents parallel computations that can be divided into pieces of arbitrary sizes, where these load fractions can be processed independently with no precedence constraints. In recent years, divisible load theory (DLT) has been widely studied for a wide range of applications in heterogeneous computing, such as image and signal processing, database applications and linear algebra [11, 12].

The problem of scheduling DL applications can generally be viewed as two-fold. Firstly, it is decided how many "load units" has to be sent to each device to achieve a balanced load distribution. Secondly, in order to reduce the impact of inevitable delays when distributing and retrieving the load, each individual load (assigned to a device) needs to be sub-partitioned into many smaller chunks, in order to: *(i)* overlap computation and communication; *(ii)* efficiently use the

communication links in desktop systems typically with bidirectional asymmetric bandwidth; *(iii)* respect the amount of supported concurrency, and *(iv)* fit into the limited device memory. In DLT, this organization is usually referred as *multi-installment (multi-round) processing* [12].

Although several authors have already studied multi-installment divisible load scheduling (DLS) in heterogeneous star networks [2], with limited memory [3, 5], and results collection [7], we show in this paper that the considered restrictions and assumptions may be realistic for some particular applications, but certainly not for all of them. These studies considered: an one-port communication model with symmetric bandwidth; when dealing with limited memory, the main focus is on fitting the input load into the finite memory size; they derive the closed-form solutions or optimal DLS algorithms by modeling computation and communication times by linear or affine functions of the number of chunks.

We naturally target a heterogeneous star (master-worker) networks, due to the basic architectural principle of desktop systems, which positions the CPU (host, master) to be responsible for controlling and orchestrating the operation of all the processing devices (workers), namely GPUs. The system heterogeneity is described with different processing speeds for each worker, and different bandwidths of master-worker communication links. Precisely, each link is modeled as an asynchronous bidirectional full-duplex communication channel with asymmetric bandwidth in different transfer directions. In contrast to the previous works dealing with limited memory [3, 5], we consider the real application behavior, that generally requires additional memory space to be allocated during the processing. Moreover, we do not make any assumptions, but we rather model computation and communication via continuous functions of the number of chunks, constructed from the real application execution. The works presented in [4, 10] also model computation through continuous functions, but do not consider either communication or multi-installment scheduling. To the best of our knowledge there is only one publication dealing with DLS problems in desktop systems [1], but for a specific application and based on affine cost models.

2 System Model and Problem Formulation

Let $\mathcal{D} = (\mathcal{A}, \mathcal{H}, \psi_t, \psi_w, \sigma_\iota, \sigma_o, \mu_\iota, \mu_w, \mu_o)$ be a DLS system, where the divisible load $\mathcal{A}$ is to be distributed and processed on a heterogeneous star network $\mathcal{H} = (P, B, E)$. Due to employment of master cores for execution, a set of $k + m$ processing devices is defined as $P = P_M \cup P_W$, where $P_M = \{p_1, ..., p_k\}$ is a set of k cores on the CPU master (positioned at the center of the star), and $P_W = \{p_{k+1}, ..., p_{m+k}\}$ is a set of m 'distant workers". Unless stated otherwise, we will use the term "distant worker" to designate a processing device, such as GPU, connected to the master via a communication link. $E = \{e_{k+1}, ..., e_{m+k}\}$ is a set of m links that connect the master to the P_W distant workers (locally, to perform load execution on the master cores P_M no extra communication costs are considered). $B = \{b_{k+1}, ..., b_{m+k}\}$ is the set of parameters describing the available memory at each distant worker.

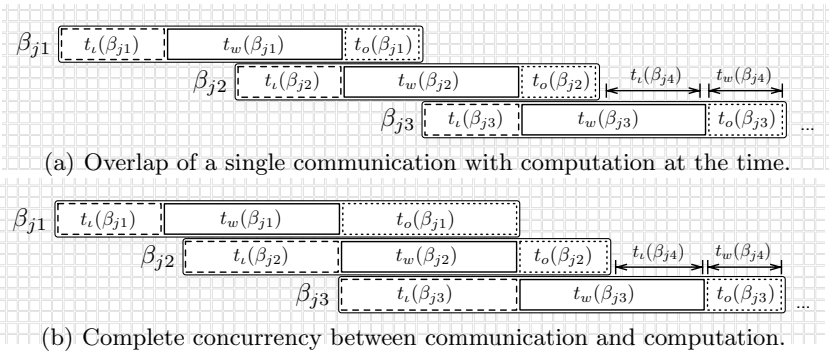
Initially, the total load N of the application $\mathcal{A}$ is stored at the master, which can be split into load fractions of an arbitrary size x. In contrast to the usual DLT practice to model computation/communication time with linear or affine functions of the load size x, we describe these relations with performance functions dynamically built during the application execution. In detail, for each load fraction x processed on a device or communicated over a link for a certain time t, we calculate its relative performance x/t in order to construct the function $f : \mathbb{N} \to \mathbb{R}_+$ which is continuously extended by piece-wise linear approximation to a performance function $g : \mathbb{R}_+ \to \mathbb{R}_+$ ($f(x) = g(x), \forall x \in \mathbb{N}$), such as in [4]. Therefore, those performance models are much more realistic in capturing the behavior of applications and the characteristics of complex heterogeneous systems [10]. Hence, $\psi_w(x)$ models relative computation performance of each P device as a function of the load size x. Bidirectional full-duplex asymmetric bandwidth of each link from E is modeled with $\sigma_\iota(x)$ and $\sigma_o(x)$ functions (where index ι reflects the communication direction from the master to a distant worker, and index o from a distant worker to the master). Dedicated links for each master-distant worker pair allow modeling of total relative performance with $\psi_t(x)$ function, calculated as ratio between the load size x and the total time taken to distribute and process the load and to return the results.

Current research in limited memory DLS [3, 5] mainly focuses on fitting the input load into the worker's memory (or input buffer) without considering the additional memory allocated during processing. Although for traditional distributed systems (with CPUs) this requirement is usually neglected, due to possible allocation in virtual memory address space, this can not be assumed for accelerators in desktop systems, such as GPUs. Therefore, we characterize the application by input, output and execution memory requirement functions, $\mu_\iota(x)$, $\mu_o(x)$, and $\mu_w(x, P)$, respectively. We define the execution memory requirement $\mu_w(x, P)$ as a function of load size x and device type P, to express the high level of heterogeneity in modern desktop systems where different implementations of the same problem might have different memory requirements.

The first part of the problem is how to divide the total load N into fractions $\alpha = \{\alpha_1, ..., \alpha_k, \alpha_{k+1}, ...\alpha_{m+k}\}$ to be simultaneously processed on each master core and distant worker $p_1, ..., p_k, p_{k+1}, ..., p_{k+m}$, such that the load distribution α is as balanced as possible. We adopt herein the results of the research conducted in [10] for the case without any communication modeling, to our communication-aware total performance functions $\psi_t(x)$. Hence, the optimal load distribution lies on a straight line that passes through the origin of the coordinating system and intersects $\psi_t(x)$ performance functions, such that:

$$\frac{\alpha_1}{\psi_{t_1}(\alpha_1)} = \cdots = \frac{\alpha_k}{\psi_{t_k}(\alpha_k)} = \cdots = \frac{\alpha_{m+k}}{\psi_{t_{m+k}}(\alpha_{m+k})}; \quad \sum_{i=1}^{m+k} \alpha_i = \mathrm{N} \tag{1}$$

We tackle herein the second part of the DLS problem: how to sub-partition a given load fraction $\{\alpha_i\}_{i=k+1}^{m+k}$ in terms of number of chunks and number of sub-load distributions at the distant worker p_i according to: *(i)* relative performance models of computation, ψ_{w_i}, and asymmetric full-duplex network links, $\sigma_{\iota_i}, \sigma_{o_i}$; *(ii)* amount of concurrency supported by the p_i; and *(iii)* limited worker's memory b_i, such that the processing is finished in the shortest time.



(a) Overlap of a single communication with computation at the time.

(b) Complete concurrency between communication and computation.

Fig. 1. Examples of different amounts of overlapping concurrency.

3 Algorithm for Device-level Multi-installment with Multi-distributions

As referred in the previous section, current multi-installment DLS studies [3, 5] consider the one-port communication model, where the load fractions are sub-partitioned into smaller chunks to allow earlier activation of workers. Thus, the problem is how to interleave those chunks (in rounds) between the workers to reduce the total application processing time by overlapping computation at one worker with communication between the master and the next worker. In this paper, we consider asymmetric full-duplex model which allows the master to simultaneously communicate with distant workers. Hence, the problem investigated herein is how to sub-partition the given load α_i to reduce the processing time at a single distant worker level by overlapping worker's sub-partitions. We allow the load α_i to be partitioned into sub-load distributions $\beta = \{\beta_j\}, 1 \leq j \leq |\beta|$, each of them consisting of sub-load fractions $\beta_j = \{\beta_{j,l}\}, 1 \leq l \leq |\beta_j|$, such that the total sum of load fractions is equal to α_i. The DLS procedure requires in-order scheduling of β_j distributions, and overlapped execution of $\beta_{j,l}$ fractions in each of them, which means we define a *single device multi-installment execution space with multi-distributions*. For limited memory systems, each β_j distribution must satisfy an additional requirement that the sum of input, output and execution memory requirements of each sub-fraction $\beta_{j,l}$ must fit into the available memory size b_i of the distant worker p_i. In fact, each sub-distribution may consume the whole amount of available memory, which is released between consecutive and independent β_j distributions.

Fig. 1 depicts the optimal overlap of three load fractions in a single load distribution for examples with different amounts of concurrency. The linear programming formulation of the problem in Fig. 1(a) states the necessary conditions for optimal overlap between subsequent load fractions in a β_j sub-distribution, where $t_\delta(\beta_j)$ represents the total processing time of β_j. We define the sub-distribution maximum relative performance, $\psi_\delta(\beta_j)$, as the optimality criterion, due to ability not only to select the distribution that efficiently uses both computation power of the distant worker p_i, ψ_{w_i}, and asymmetric bandwidth of the

e_i network link, σ_{ι_i} and σ_{o_i}, but also to minimize the impact of the intrinsic communication overhead in the first, $\beta_{j,1}/\sigma_{\iota_i}(\beta_{j,1})$, and the last load fraction, $\beta_{j,|\beta_j|}/\sigma_{o_i}(\beta_{j,|\beta_j|})$, to the overall β_j execution time.

MAXIMIZE $\psi_\delta(\beta_j)=\frac{\sum_{l=1}^{|\beta_j|}\beta_{j,l}}{t_\delta(\beta_j)}$ SUBJECT TO:

$$\sum_{l=1}^{|\beta_j|}\beta_{j,l}\leq\alpha_i;\ \ \sum_{l=1}^{|\beta_j|}\left(\mu_\iota(\beta_{j,l})+\mu_w(\beta_{j,l},p_i)+\mu_o(\beta_{j,l})\right)\leq b_i \tag{2}$$

$$\frac{\beta_{j,1}}{\psi_{w_i}(\beta_{j,1})}\geq\frac{\beta_{j,2}}{\sigma_{\iota_i}(\beta_{j,2})};\ \ \frac{\beta_{j,3}}{\psi_{w_i}(\beta_{j,3})}\geq\frac{\beta_{j,2}}{\sigma_{o_i}(\beta_{j,2})} \tag{3}$$

$$\frac{\beta_{j,2}}{\psi_{w_i}(\beta_{j,2})}\geq\frac{\beta_{j,1}}{\sigma_{o_i}(\beta_{j,1})}+\frac{\beta_{j,3}}{\sigma_{\iota_i}(\beta_{j,3})} \tag{4}$$

$$\frac{\beta_{j,l}}{\sigma_{\iota_i}(\beta_{j,l})}\leq\frac{\beta_{j,l-1}}{\psi_{w_i}(\beta_{j,l-1})}-\frac{\beta_{j,l-2}}{\sigma_{o_i}(\beta_{j,l-2})};\ \ \frac{\beta_{j,l}}{\psi_{w_i}(\beta_{j,l})}\geq\frac{\beta_{j,l-1}}{\sigma_{o_i}(\beta_{j,l-1})};\ \ \forall l\in\{4,|\beta_j|\} \tag{5}$$

It can be observed that the number of sub-distributions and load fractions depend not only on the system capabilities and concurrency amount, but also on the application's computation and communication characteristics. In the general case, it may not even be possible to satisfy all the above-mentioned conditions. Hence, we propose herein an algorithm that finds a sub-optimal β distribution from a closed set $\beta^*=\{\beta^{(n)}\}_{n=1}^{|\beta^*|}$, where $\sum_{j=1}^{|\beta^{(n)}|}\sum_{l=1}^{|\beta_j^{(n)}|}\beta_{j,l}^{(n)}=\alpha_i,\forall n\in\{1,|\beta^*|\}$, and $\sum_{l=1}^{|\beta_j^{(n)}|}(\mu_\iota(\beta_{j,l}^{(n)})+\mu_w(\beta_{j,l}^{(n)},p_i)+\mu_o(\beta_{j,l}^{(n)}))\leq b_i,\forall n,j$, such that:

$$\beta=\beta^{(r)};\ \ \psi_\tau(\beta^{(r)})=\max\{\psi_\tau(\beta^{(n)})\}_{n=1}^{|\beta^*|};\ \ \psi_\tau(\beta^{(n)})=\alpha_i/t_\tau(\beta^{(n)}), \tag{6}$$

where $t_\tau(\beta^{(n)})$ is the total processing time of a $\beta^{(n)}$ distribution on the distant worker, and $\psi_\tau(\beta^{(n)})$ its total relative performance. Therefore, in order to construct the set β^*, we firstly determine the initial optimal load distribution with three load fractions satisfying (2) and (4). Due to the space limitations, only the main algorithm steps are present herein, more details can be found in [8].

Step 1. *Determination of the initial optimal distribution with three load fractions.* The algorithm firstly determines the optimal solution search space limits by finding minimum and maximum for $\beta_{1,1}^{(1)}$, $\beta_{1,2}^{(1)}$, and $\beta_{1,3}^{(1)}$ [8]. In case of Fig. 1(a), the process of finding the optimal initial solution requires to build interpolated curves from discreet values: *(i)* $\sigma_{\iota_i}^{(x_1)}$ from σ_{ι_i} and $x_1\in[\beta_{1,1_{\min}}^{(1)},\beta_{1,1_{\max}}^{(1)}]$; *(ii)* $\sigma_{o_i}^{(x_3)}$ from σ_{o_i} and $x_3\in[\beta_{1,3_{\min}}^{(1)},\beta_{1,3_{\max}}^{(1)}]$. Then, the optimal solution is the maximum relative performance, ψ_δ, solution that satisfies (2) and (4) and lies on a straight line passing through the coordinate system origin, such that:

$$\frac{\beta_{1,1}^{(1)}+\beta_{1,3}^{(1)}}{\sigma_{\iota_i}^{(x_1)}\left(\beta_{1,1}^{(1)}+\beta_{1,3}^{(1)}\right)}=\frac{\beta_{1,2}^{(1)}}{\psi_{w_i}\left(\beta_{1,2}^{(1)}\right)},\ \ \frac{\beta_{1,1}^{(1)}+\beta_{1,3}^{(1)}}{\sigma_{o_i}^{(x_3)}\left(\beta_{1,1}^{(1)}+\beta_{1,3}^{(1)}\right)}=\frac{\beta_{1,2}^{(1)}}{\psi_{w_i}\left(\beta_{1,2}^{(1)}\right)}. \tag{7}$$

If no optimal initial distribution is found, the algorithm continues with **Step 5**.

Step 2. *Generate additional three-fraction distributions.* In order to satisfy (3), at maximum three additional distributions are created with the optimal initial distribution, if permitted by the application characteristics and (2). Initially, $\{\beta_{1,2}^{(n)}\}_{n=2}^{4}=\beta_{1,2}^{(1)}$ ($\beta_{1,3}^{(2)}=\beta_{1,3}^{(1)}$, $\beta_{1,1}^{(3)}=\beta_{1,1}^{(1)}$) values are assigned, and the remaining loads are determined to completely overlap their computation and communication in $\{\beta_{1,2}^{(n)}\}_{n=2}^{4}$. Namely, the values lie on a straight line passing through the coordinate system origin, such that (2), (3), and:

$$\frac{\beta_{1,1}^{(2)}}{\psi_{w_i}\left(\beta_{1,1}^{(2)}\right)}=\frac{\beta_{1,2}^{(1)}}{\sigma_{\iota_i}\left(\beta_{1,2}^{(1)}\right)},\ \ \beta_{1,1}^{(2)}=\beta_{1,1}^{(4)};\ \ \frac{\beta_{1,3}^{(3)}}{\psi_{w_i}\left(\beta_{1,3}^{(3)}\right)}=\frac{\beta_{1,2}^{(1)}}{\sigma_{o_i}\left(\beta_{1,2}^{(1)}\right)},\ \ \beta_{1,3}^{(3)}=\beta_{1,3}^{(4)}. \tag{8}$$

Step 3. *Insert additional load fractions into existing sub-distributions.* Each $\{\beta^{(n)}_{|\beta^{(n)}|}\}_{n=1}^{|\beta^*|}$ sub-distribution is evaluated using the procedure from [8] to determine the possibilities to insert another load fraction into the current sub-distribution schedule, in respect to the amount of supported concurrency. Namely, each sub-distribution is assigned with two real values representing the maximum available time frames that can be allocated to the additional load fraction, i.e., $t^{(n)}_{a_\iota}$ and $t^{(n)}_{a_w}$ $(a{=}|\beta^{(n)}_{|\beta^{(n)}|}|{+}1)$, where $t^{(n)}_{a_\iota}$ and $t^{(n)}_{a_w}$ reassemble the notions from (5). Correspondingly, the next load fractions $\beta^{(n)}_{|\beta^{(n)}|,a}$ lie on a straight line passing through the origin of the coordinate system, such that (2), (5), and:

$$\frac{\beta^{(n)}_{|\beta^{(n)}|,a}}{\psi_{w_i}(\beta^{(n)}_{|\beta^{(n)}|,a})} = t^{(n)}_{a_w} \text{ or } \frac{\beta^{(n)}_{|\beta^{(n)}|,a}}{\sigma_{\iota_i}(\beta^{(n)}_{|\beta^{(n)}|,a})} = t^{(n)}_{a_\iota}. \tag{9}$$

The obtained values are inserted as the last load-fractions for current sub-distribution and an additionally created sub-distribution by duplicating current $\beta^{(n)}$ schedule. In the case that the calculated load fraction does not satisfy (2), the sub-distribution is marked as examined. This procedure is iterative, until all $\{\beta^{(n)}_{|\beta^{(n)}|}\}_{n=1}^{|\beta^*|}$ sub-distributions are examined.

Step 4. *Generate new sub-distributions by restarting.* After the insertion, in each $\{\beta^{(n)}\}_{n=1}^{|\beta^*|}$ schedule the new sub-distribution is added on which the complete procedure is repeated (from **Step 1**) for the remaining unscheduled load:

$$\alpha_i - \sum_{j=1}^{|\beta^{(n)}|}\sum_{l=1}^{|\beta^{(n)}_j|}\beta^{(n)}_{j,l}, \forall n \in \{1, |\beta^*|\} \tag{10}$$

In case that the amount of remaining load is insufficient to produce the optimal three-fraction sub-distribution, the algorithm proceeds with **Step 5**.

Step 5. *Expand all sub-distributions.* In this step, each schedule $\{\beta^{(n)}\}_{n=1}^{|\beta^*|}$ is expanded with a single sub-distribution containing a single load fraction of the size obtained by (10) in case that (2) is satisfied, or with multiple single-fraction distributions which are iteratively assigned to meet the condition from (2).

Step 6. *Select the schedule with maximum relative performance.* As previously referred, the final schedule β is selected from the β^* set according to (6).

4 Iterative Procedure for Partial Performance Modeling

The proposed approach relies on relative performance models of system resources, which are usually not known a priori. Hence, we propose an iterative procedure with two main phases (*initialization* and *iterative phase*), based on the algorithm presented in Section 3 that, at the same time, builds the models and makes scheduling decisions according to their current partial estimations.

Initialization phase begins by assigning $\alpha_i = N/(m{+}k)$ load fractions to each p_i device. For each distant worker $\{p_i\}_{i=k+1}^{m+k}$, the initial α_i load is iteratively split into sub-fractions by applying a factoring-by-two strategy. In general, this results in a single sub-distribution β^i_1 with sub-fractions calculated as:

$$\beta^i_{1,l} = \left\lceil \left(\frac{1}{2}\right)^l \alpha_i \right\rceil \tag{11}$$

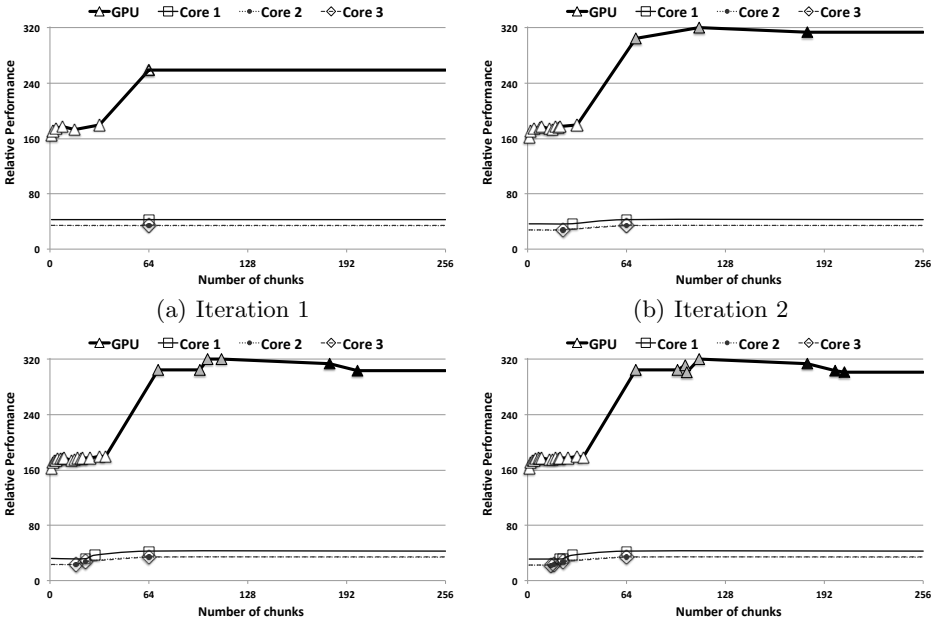
For systems with limited memory, we propose a recursive procedure that sub-partitions load-fractions from the original sub-distribution into additional sub-distributions by applying the factoring-by-two strategy, until β^i satisfies (2). After β^i schedule is processed, the initial performance models are built via piece-wise linear approximations on the real execution values [4]. Namely, ψ_{w_i}, σ_{ι_i}, σ_{o_i} and ψ_{t_i} models are constructed from the values obtained from each load-fraction, and ψ_{t_i} is further extended with the values from each sub-distribution execution.

In the *iterative phase*, the new load distribution α is calculated for each p_i device by applying (1) to the total performance models, ψ_t. For each distant worker, sub-distributions and sub-load fractions are further calculated with the algorithm presented in Section 3 by relying on partial estimations of performance models. After load processing, the device execution times are compared and if the relative differences satisfy the given accuracy, the procedure stops by marking α as the load balanced distribution. If not, the performance model estimations are updated with the newly obtained values, and the procedure restarts with the *iterative phase*, until the load balanced distribution is found.

5 Experimental Results

The proposed approach was evaluated in a real heterogeneous desktop system consisting of an Intel Core 2 Quad Q9550@2.83 GHz CPU with 12 MB L2 cache and 4 GB of DDR2 RAM, and an NVIDIA GeForce 285 GTX@1.476 GHz GPU with 1 GB of global memory. The GPU is connected to the CPU with PCI Express 2.0 x16 bus, where only a single transfer can be successfully overlapped with computation at the time. It is considered that the relative performance models were not a priori known, thus the iterative procedure was applied to the collaborative (GPU + 3 CPU Cores) execution of a real DL application performing two forward and inverse 2D batch double floating complex Fast Fourier Transforms (FFT) of size 256 times 512×512, divisible in the first dimension. We used the optimal vendor-provided FFT implementations, i.e., NVIDIA's CUFFT 3.2 for the GPU and Intel MKL 10.3 for the CPU, on Linux OpenSuse 11.3 system.

By applying the proposed algorithm, we achieved the load balanced execution in only 4 iterations, and the obtained relative per-device performance models (after each iteration) are presented in Fig. 2. In the first iteration, the total load is equally partitioned among devices, i.e., $\alpha = \{\alpha_i = 64\}_{i=1}^{4}$. The GPU load is further sub-partitioned with the factoring-by-two strategy resulting in $\beta_1 = \{32, 16, 8, 4, 2, 1, 1\}$ distribution, which load fractions are executed in overlapped fashion. As shown in Fig. 2(a), we model the GPU total performance with 7 points after a single application run, i.e., 6 points are obtained at the level of each load-faction, and the final point is the performance of a complete β_1 schedule. Repeated load fractions or sub-distributions are considered as accuracy points, which do not contribute to the overall number of points, but improve the model accuracy (in this case, one accuracy point is obtained for the load fraction size of 1). By applying factoring-by-two strategy, we obtained the speed up of about 1.4 comparing to a single-fraction non-overlapped execution of the $\alpha_i = 64$ load.



(a) Iteration 1 (b) Iteration 2

(c) Iteration 3 (d) Iteration 4

Fig. 2. Total performance models ψ_t obtained for two forward and inverse double complex 2D batch CUFFT kernels of size $x \times 512 \times 512$.

According to the current ψ_t models, the load distribution for the second integration is calculated as $\alpha = \{181(\text{GPU}), 29(\text{Core 1}), 23(\text{Core 2}), 23(\text{Core 3})\}$. The GPU load is sub-fractioned according to the algorithm presented in Section 3, by relying onto the initial partial estimations of ψ_w, σ_i and σ_o, resulting in the distribution vector: $\beta = \{\beta_1 = \{8, 14, 18, 20, 9, 1\}, \beta_2 = \{8, 14, 18, 20, 21, 21, 9\}\}$. After the second iteration, we model the GPU performance with 15 approximation points (see Fig. 2(b)), including 3 points obtained for β_1 and β_2 sub-distributions and the overall β schedule. In terms of performance, with our algorithm we outperformed the optimal single load CUFFT execution for about 4.6 times in this iteration. The calculated distributions for the third iteration are: $\alpha = \{199, 23, 17, 17\}$, and $\beta = \{\{14, 26, 36, 16, 7, 3\}, \{14, 26, 36, 16, 5\}\}$ for the GPU. As presented in Fig. 2(c), the GPU performance is modeled with 23 points after the third iteration, resulting in 4.5 speed-up comparing to the CUFFT single load execution. Finally, with the obtained distribution for the fourth iteration ($\alpha = \{205, 21, 15, 15\}$, and $\beta = \{\{14, 26, 36, 16, 7, 3\}, \{14, 26, 36, 16, 7, 3, 1\}\}$ the load balancing is achieved across all 4 devices, outperforming the single load non-overlapped CUFFT execution for about 4.3 times.

In total, by using the proposed approach, the GPU performance is modeled with 53 points in only 4 application runs, i.e., 25 approximation points (see Fig 2(d) and 28 accuracy points. In average, we obtain the speed up of about 4.5 comparing to the direct use of the optimal CUFFT library.

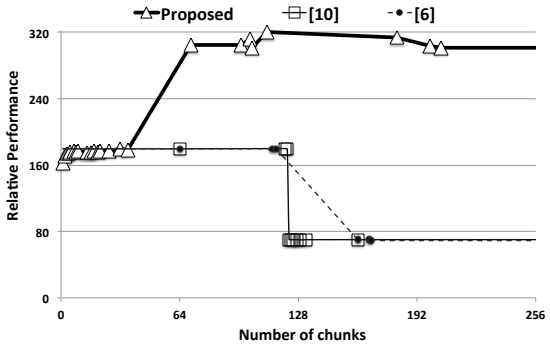


Fig. 3. Comparison of obtained performance models and relative performance.

5.1 Comparison with the state of the art approaches

In order to provide a better insight on the efficiency of the proposed algorithm, we conducted the comparison with two iterative approaches to achieve load balancing that rely on the functional performance models, [6] and [10]. Albeit those approaches are developed without any communication awareness, they can be straightforwardly applied to our communication-aware total performance curves ψ_t. The performance models in [6] are represented as constants obtained from the last application run, whereas the algorithm from [10] deals with the complete functional performance curves when deciding on load balancing. Figure 3 presents the comparison of obtained ψ_t total GPU performance models when executing the above-mentioned DL application for a situation when the load balancing is achieved (if possible). By using the approach in [10], it takes 10 iterations to converge to a steady load distribution, but even then the obtained distribution is not load balanced, due to the refinement procedure applied to the performance curve with instantaneous change in the relative performance. On the other hand, the algorithm presented in [6] will neither achieve load balancing nor converge to the final distribution, as after 8 iterations it arrives to a state referred as "ping-pong" effect [4], due to its unawareness of a complete shape of the performance curve.

For both approaches, the total number of approximation points obtained for GPU performance is equal to the number of iterations, i.e., 10 for approach in [10], and 8 for approach in [6] (where 6 points are actually contributing, and the last two are repeated), comparing to 53 points obtained with our approach. Considering the time taken to completely perform the iterative procedure, the approach in [10] requires 4.3 times more time to determine the steady state distribution, whereas the algorithm from [6] takes about 3.2 times more time to arrive to a "ping-pong" state. In terms of performance, in the final iteration, our load balanced distribution is capable of achieving more than 2 times better performance comparing to a steady state distribution from [10] (when executing the complete problem in the same execution environment), and about 2.2 times better performance than the best distribution found with [6].

6 Conclusions

This paper proposes, for the first time, an algorithm for scheduling discretely divisible applications in heterogeneous desktop systems with limited memory by considering realistic performance models of computation and bidirectional asymmetric full-duplex communication links. This algorithm achieves device level multi-installment processing with multi-distributions to allow efficient overlap of computation and communication. The presented approach was experimentally evaluated for an FFT computation using optimized libraries on a GPU and 3 CPU cores. The obtained results show its capability to outperform what is thought to be the optimal implementation for about 4 times, whereas the current state of the art approaches were incapable of determining the load balanced distribution. Moreover, by employing the proposed algorithm not only more accurate performance models are constructed significantly faster, but also the overall application performance is improved.

Acknowledgments

This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and a fellowship SFRH/BD/44568/2008.

References

1. Barlas, G.D., Hassan, A., Jundi, Y.A.: An Analytical Approach to the Design of Parallel Block Cipher Encryption/Decryption: A CPU/GPU Case Study. In: PDP. pp. 247–251 (2011)
2. Beaumont, O., et al.: Scheduling divisible loads on star and tree networks: results and open problems. IEEE Trans. Parallel Distributed Systems 16, 2005 (2003)
3. Berlińska, J., Drozdowski, M.: Heuristics for multi-round divisible loads scheduling with limited memory. Parallel Comput. 36, 199–211 (2010)
4. Clarke, D., Lastovetsky, A., Rychkov, V.: Dynamic load balancing of parallel computational iterative routines on platforms with memory heterogeneity. In: HeteroPar 2010 (2010)
5. Drozdowski, M., Lawenda, M.: A new model of multi-installment divisible loads processing in systems with limited memory. In: PPAM'07. pp. 1009–1018. Springer-Verlag (2008)
6. Galindo, I., Almeida, F., Badía-Contelles, J.M.: Dynamic load balancing on dedicated heterogeneous systems. In: PVM/MPI. pp. 64–74 (2008)
7. Ghatpande, A., Nakazato, H., Watanabe, H., Beaumont, O.: Divisible load scheduling with result collection on heterogeneous systems. In: IPDPS. pp. 1–8 (2008)
8. Ilic, A., Sousa, L.: Algorithm for divisible load scheduling on heterogeneous systems with realistic performance models. Tech. rep., INESC-ID (May 2011)
9. Ilic, A., Sousa, L.: Collaborative execution environment for heterogeneous parallel systems. In: APDCM/IPDPS 2010 (2010)
10. Lastovetsky, A., Reddy, R.: Data partitioning with a functional performance model of heterogeneous processors. Int. J. High Perform. Comput. Appl. 21, 76–90 (2007)
11. Shokripour, A., Othman, M.: Survey on divisible load theory and its applications. In: ICIME 2009. pp. 300 –304 (2009)
12. Veeravalli, B., Ghose, D., Robertazzi, T.G.: Divisible load theory: A new paradigm for load scheduling in distributed systems. Cluster Computing 6, 7–17 (2003)