HAL
open science

Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems

Maxime Gonthier, Loris Marchal, Samuel Thibault

To cite this version:

Maxime Gonthier, Loris Marchal, Samuel Thibault. Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems. IPDPS 2022 - 36th IEEE International Parallel & Distributed Processing Symposium, May 2022, Lyon, France. pp.1-11, 10.1109/IPDPS53621.2022.00073 . hal-03552243

HAL Id: hal-03552243

https://inria.hal.science/hal-03552243

Submitted on 3 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems

Maxime Gonthier
LaBRI, LIP, Inria & ENS-Lyon
maxime.gonthier@ens-lyon.fr

Loris Marchal
LIP, CNRS, ENS-Lyon, Inria
loris.marchal@ens-lyon.fr

Samuel Thibault
LaBRI, CNRS, Inria & Univ. Bordeaux
samuel.thibault@u-bordeaux.fr

***Abstract*—The use of accelerators such as GPUs has become mainstream to achieve high performance on modern computing systems. GPUs come with their own (limited) memory and are connected to the main memory of the machine through a bus (with limited bandwidth). When a computation is started on a GPU, the corresponding data needs to be transferred to the GPU before the computation starts. Such data movements may become a bottleneck for performance, especially when several GPUs have to share the communication bus.**

Task-based runtime schedulers have emerged as a convenient and efficient way to use such heterogeneous platforms. When processing an application, the scheduler has the knowledge of all tasks available for processing on a GPU, as well as their input data dependencies. Hence, it is able to choose which task to allocate to which GPU and to reorder tasks so as to minimize data movements. We focus on this problem of *partitioning* and *ordering* tasks that share some of their input data. We present a novel dynamic strategy based on data selection to efficiently allocate tasks to GPUs and a custom eviction policy, and compare them to existing strategies using either a well-known graph partitioner or standard scheduling techniques in runtime systems. We also improved an offline scheduler recently proposed for a single GPU, by adding load balancing and task stealing capabilities. All strategies have been implemented on top of the StarPU runtime, and we show that our dynamic strategy achieves better performance when scheduling tasks on multiple GPUs with limited memory.

I. Introduction

High-performance computing applications, such as simulation for aeronautics, material resistance, or seismology, keep demanding increasingly intense computation power on ever-growing sets of data. The past decade has been marked by a trend to leverage GPUs in addition to CPUs, to achieve unforeseen computation speed as well as energy requirements efficiency. Taking the most benefit from such combination of CPUs and GPUs is however very challenging, since they exhibit different computational efficiencies, and GPUs embed their own dedicated high-bandwidth memory, which requires transferring data between CPUs and GPUs. To tackle this concern, it has become very common to use the task-based programming paradigm, i.e. to express the application computation as a Directed Acyclic Graph (DAG), and let a dynamic runtime system such as OmpSs [1], PaRSEC [2], or StarPU [3] manage the execution of the task graph over such distributed and heterogeneous platforms. The burden is thus offloaded from the application programmer to the runtime system, in the form of a task scheduling problem.

The challenge is not only that such platforms are composed of largely heterogeneous resources, but also that the memory embedded in GPUs is relatively limited, and the bus that connects them to the main memory has a very limited bandwidth. The runtime scheduler thus not only has to care for task acceleration, it also has to take into account the movement of data within the system. This means that it must carefully decide the task mapping on GPUs according to data locality, as well as the ordering of the tasks itself, to privilege the temporal locality of data, thus favoring data reuse and saving duplicate data transfers. It is also essential to trigger data transfers ahead of task execution (data prefetches) so that they can be overlapped, i.e. they proceed during the execution of the previous tasks. Last but not least, when the embedded memory of the GPU is full, the runtime has to carefully decide which data should be evicted from it, to make room for further data.

We focus in this paper on the problem of **partitioning and scheduling a set of tasks on one and multiple GPUs with limited memory, where tasks share some of their input data but are otherwise independent**, as well as managing data movements (loads and evictions). More precisely, we want to determine the order in which tasks must be processed on each GPU to optimize for data reuse as well as maximize overlap between computations and data movements. Our objective is twofold: (i) we partition the work on each GPU to reach a good load balance, and (ii) we want to minimize the total amount of data transferred to the GPUs for the processing of all tasks with a constraint on the memory size. We start focusing on independent tasks sharing input data because when using usual dynamic runtime schedulers, the scheduler is exposed at a given time to a fairly large subset of tasks which are independent of each others. This is in particular the case with linear algebra workflows, such as the matrix multiplication: except possibly at the very beginning or very end of the computation, a large set of tasks is available for scheduling. Thus, solving the bi-objective optimization problem for the currently available tasks can lead to a large reduction in data transfers and hence a performance increase, which is ultimately our goal.

In this paper, we make the following contributions:

- We provide a formal model of the bi-objective optimization problem, and observe the problem to be NP-complete (Section III).

- We review and adapt a (hyper-)graph partitioning method from the literature for this problem (Section IV-B).
- We adapt a previously-proposed scheduling heuristic, HFP, to the multi-GPU case (Section IV-C).
- We propose a new heuristic based on performing as much computations as possible with data at hand as well as an eviction policy focused on finding the least used data in the future (Section IV-D).
- We implement all three heuristics into the StarPU runtime and study the performance (amount of data transfers and total processing time) obtained on a 2D-blocked matrix multiplication, a 3D matrix multiplication, task from the Cholesky decomposition, as well as a sparse 2D-blocked matrix multiplication (Section V). Overall, our evaluation shows that our heuristic generally surpasses previous strategies, in particular in the most constrained situations.

Note that while we focus our experimental validation on GPUs, the optimization problem studied in this paper is not specific to the use of such accelerators: it appears as soon as tasks sharing data must be processed on a system with limited memory and bandwidth. For example, it is also relevant for a computer made of several CPUs with restricted private memory, and limited bandwidth for the communication between memories and disk.

For simplicity, we do not consider the output data of tasks: In the case of linear algebra for instance, the output data is most often much smaller than the input data and can be transferred concurrently with data input. Data output is then not the driving constraint for efficient execution. Our model could however easily be extended to integrate task output.

II. Related Work

We describe here the related work on the various topics covered by this study.

a) Mapping and Scheduling Tasks Sharing Data: The problem of tasks sharing input files has been extensively studied in scheduling for distributed platforms. In particular Giersch et al. [4] studied how to allocate and schedule tasks sharing input files on a distributed platform, when the communication between the server holding the input files and the workers is limited. Senger et al. [5] proposed a hierarchical strategy for data distribution in order to improve scalability. Kaya et al. refined the problem by considering that input files are initially distributed on the platform, but may also be transferred through the network if required. They notably proposed heuristics using hypergraph partitioning [6], which inspired one of the strategy of the present study.

b) Data Locality: There have been plenty of studies on data locality to improve performance, from the seminal work of Hong & Kung [7], many of them targeting linear algebra (see [8], [9] for recent works). The work of Yao et al. [10] is very close to our problem: they optimize the scheduling of independent tasks sharing input data, but target multicore CPUs. Our hMETIS+R strategy (see below) elaborates on ideas presented in their paper.

c) Scheduling in Task Based Runtime Systems: As outlined in the introduction, runtime systems like OmpSs [1], PaRSEC [2], or StarPU [3] are increasingly popular to cope with the complexity of modern computing platforms. In the XKaapi runtime system which implements a work-stealing scheduler, efforts have been made to favor data locality [11] by implementing and extending ideas from theoretical studies on data locality for work stealing [12]. On the contrary, in the default DMDAR scheduler of StarPU (presented below), tasks are scheduled where there are expected to complete at the earliest, which takes data transfers into account. In the present study, we consider data locality first, using work-stealing or other load balancing techniques as a secondary step.

III. Problem Modeling

We consider the problem of scheduling independent tasks on K GPUs, denoted by $\mathrm{GPU}_1, \ldots, \mathrm{GPU}_K$. As proposed in previous work [13], tasks sharing their input data can be modeled as a bipartite graph $G = (\mathbb{T} \cup \mathbb{D}, E)$. The vertices of this graph are on one side the tasks $\mathbb{T} = \{T_1, \ldots, T_m\}$ and on the other side the data $\mathbb{D} = \{D_1, \ldots, D_n\}$. An edge connects a task T_i and a data D_j if task T_i requires D_j as input data. For the sake of simplicity, we denote by $\mathcal{D}(T_i) = \{D_j \text{ s.t. } (T_i, D_j) \in E\}$ the set of input data for task T_i. We here consider that all data have the same size. Each of the K GPUs is equipped with a memory of limited size, which may contain at most M data simultaneously. Initially, all input data are stored in the main memory of the machine. During the processing of a task T_i on GPU_k, all its inputs $\mathcal{D}(T_i)$ must be in the memory of GPU_k. Note that as presented above, we here do not consider the data output of tasks.

Each of the m tasks must be processed on some of the K GPUs. Our goal is to determine both **how to partition** the task set to the GPUs and in **which order** to process them on each GPU. As explained above, our objective is also to come up with a schedule with few data movement, as they can largely impact the overall processing time. Hence we also need to detail **when each data must be loaded or evicted** from the memory of each GPUs.

We consider here that all GPUs are connected to the main memory of the machine (which originally contains all input data) through a shared bus (see Figure 2). The bounded bandwidth of this bus is the reason why we aim at restricting the amount of data movement between the main memory and the GPUs.

We now define more formally the allocation of the tasks to the GPU and their schedule. We denote by $\sigma(k, i)$ the ith task processed on GPU_k, and by $\mathcal{V}(k, i)$ the set of data to be evicted from the memory of GPU_k before the processing of this i^{th} task. We also let nb_k be the number of tasks allocated to GPU_k. On GPU_k, the schedule is made of a succession of nb_k steps, each step being composed of the following three stages (in this order):

1) All data in $\mathcal{V}(k, i)$ are evicted (unloaded) from the memory of GPU_k;

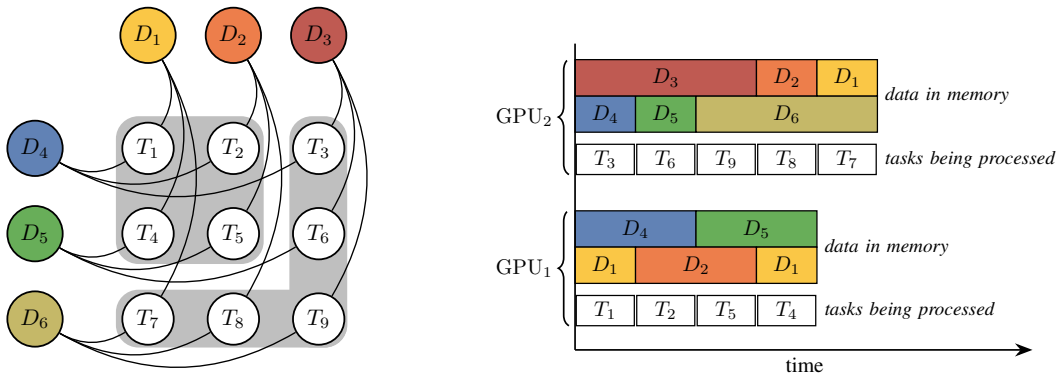

Figure 1: Small example with 9 tasks with 2D grid dependencies. The graph of input data dependencies is shown on the left, together with the task partition among GPUs. A possible schedule is described on the right. The bound on the maximum data on each GPU memory is $M = 2$. GPU$_1$ processes tasks $T_1, T_2, T_5, T4$ (in this order), and data D_1 has to be loaded twice. GPU$_2$ processes tasks T_3, T_6, T_9, T_8, T_7 in this order to avoid multiple loads of the same data. Overall, the total amount of data movement (i.e., data loads) is 11.

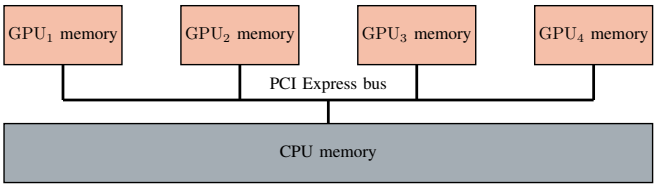

Figure 2: Platform topology.

2) The input data in $\mathcal{D}(T_{\sigma(k,i)})$ that are not yet in memory are loaded in the memory of GPU$_k$;
3) Task $T_{\sigma(k,i)}$ is processed on GPU$_k$.

An example is shown in Figure 1. This example illustrates that input data are loaded in the GPU memories as late as possible: loading them earlier would be pointless and possibly trigger more data movements. In real computing systems, a pre-fetch is usually designed to load data a bit earlier so as to avoid waiting for unavailable data, however, for the sake of simplicity, we do not consider this in our model: if needed, we may simply book part of our GPU memory for the pre-fetch mechanism.

Using the previous definition, we define the *live data* $L(k,i)$ as the data in memory of GPU$_k$ during the computation of $T_{\sigma(k,i)}$, which can be defined recursively:

$$L(k,i) = \begin{cases} \mathcal{D}(T_{\sigma(k,1)}) & \text{if } i = 1 \\ \Big(L(k,i-1) \backslash \mathcal{V}(k,i)\Big) \cup \mathcal{D}(T_{\sigma(k,i)}) & \text{otherwise} \end{cases}$$

This is explained as follows: when processing its very first task, the memory of a GPU contains only the inputs of this task. When processing some task $T_j = T_{\sigma(k,i)}$ at step i, first some data are possibly evicted from the memory of GPU$_k$ (that is $\mathcal{V}(k,i)$), then the missing inputs of task T_j are loaded.

As noted above, each GPU has a bounded memory so it can only accommodate M distinct input data (we recall that all data have the same size). This can be expressed as

$$\forall k = 1, \ldots, K, \forall i = 1, \ldots, nb_k, \qquad |L(k,i)| \leq M.$$

Our objective is both to ensure a good load balancing and to minimize the amount of data movement:

Objective 1: Load Balancing. We assume that all tasks have the same processing time on any GPU. Thus, load-balancing the work on each GPU amounts to minimizing the maximum number of tasks on any GPU:

$$\textbf{Obj. 1}: \quad \textit{minimize} \max_k nb_k$$

Objective 2: Data Movement. The second objective is to limit the amount of data movement, that is, to minimize the number of *load* operations from the main memory to the memory of the GPUs: we consider that data are not modified so no *store* operation occurs when evicting a data from a GPU memory. With the natural assumption that no input for task $T_{\sigma(k,i)}$ is evicted from GPU$_k$ right before its processing ($\mathcal{V}(k,i) \cap \mathcal{D}(T_{\sigma(k,i)}) = \emptyset$), the number of loads on GPU$_k$ can be computed as follows:

$$\#Loads_k = \sum_i \Big|\mathcal{D}\left(T_{\sigma(k,i)}\right) \backslash L(k,i-1)\Big|$$

Then, our objective is simply to minimize the total number of loads:

$$\textbf{Obj. 2}: \quad \textit{minimize} \sum_k \#Loads_k$$

In prior work, the special case with a single GPU have been studied [14], and it has been shown that given a schedule σ, it is possible to derive an optimal eviction policy $\mathcal{V}$ by following Belady's rule [15]: always evict the data whose next usage is the furthest in the future. This rule can be extended to the multi-GPU case: once tasks have been partitioned among GPUs and ordered for computation, that is, once σ is set, we may compute the optimal eviction scheme for each GPU by applying Belady's rule. Hence our objective is only to find a schedule σ of the tasks on the GPUs. The decision version of the bi-objective problem is then expressed as follows.

Definition 1 (Bi-Obj-Multi-GPU-Task-Scheduling): Given a number K of GPUs, m tasks sharing their inputs according to a bipartite graph G, and two bounds W and C, is there a schedule σ such that $\max_k nb_k \leq W$ and $\sum_k \#Loads_k \leq C$?

A previous study of the single-GPU case [14] proved that ordering tasks to minimize the data movement is NP-complete. As this sub-problem is contained in the more general problem presented here, this proves the complexity of the bi-objective problem.

Theorem 1: The Bi-Obj-Multi-GPU-Task-Scheduling problem is NP-complete.

Note that the previous model can easily be extended to heterogeneous tasks (with different processing times) and heterogeneous data (with different sizes).

IV. Algorithms

In this section, we present the various algorithmic solutions to solve the partitioning and scheduling problem presented above. Some of these methods first solve the partitioning problem, and then schedule the tasks on each GPU (Sections IV-A and IV-B) while others tackle both problems simultaneously (Sections IV-C and IV-D).

A. Dynamic scheduler of StarPU: DMDAR

DMDA or "Deque Model Data Aware" (Algorithm 1) is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing units in the StarPU runtime [16] (also called tmdp-pr). It computes the expected completion time $C_k(T_i)$ of the first task T_i in the queue on each GPU_k, based on a prediction of the time for transferring the data to the GPU (or communication time) $comm$ and of the task computation time $comp$:

$$C_k(T_i) = \sum_{\substack{j \in \mathcal{D}(T_i) \\ D_k \notin InMem(k)}} comm_k(D_j) + comp_k(T_i) \quad (1)$$

Note that the data transfer time is counted only if the data is not already in the memory of GPU_k. Then, the task is allocated on the GPU where its completion time is minimal.

Tasks are first allocated to GPUs following their order of submission. Here, we consider the DMDAR variant, which includes an additional *Ready* strategy (Algorithm 2): on each GPU, tasks are reordered at runtime in order to favor tasks with the most input data already loaded into memory[1].

Algorithm 1 Deque Model Data Aware heuristic (DMDA)

For each GPU_k, $InMem(k) \leftarrow \emptyset$
while all tasks have not been allocated **do**
  $T_i \leftarrow pop(\mathbb{T})$
  For each GPU_k, compute $C_k(T_i)$ using Eq. 1
  Select k such that $C_k(T_i)$ is minimal
  Allocate T_i on GPU_k
  for each $D_i \in \mathcal{D}(T_i)$ **do**
    Request data prefetch for D_j on GPU_k
    Add D_i to $InMem(k)$

Algorithm 2 Ready reordering heuristic on GPU_k

Input: List L of tasks allocated on GPU_k
while $L \neq \emptyset$ **do**
  Search first $T \in L$ requiring the fewest data transfers
  Wait for all data in $\mathcal{D}(T)$ to be in GPU_k memory
  Start processing T

B. Using (hyper-)graph partitioning

As outlined in [10], a graph partitioner is a very suitable tool to decompose the set of tasks sharing input data into several subsets of similar size while minimizing the number of common data among subsets. Each subset is then allocated to a GPU, and tasks within the subset are scheduled to further increase data locality. The fact that subsets have similar sizes ensures the load balancing among GPUs, while the minimization of common data reduces the amount of data that must be sent to several GPUs.

In [10], the authors model data reuse through a graph whose vertices are tasks, and edges between two tasks are weighted by the amount of shared input data between these two tasks. Then, they use the METIS [17] graph partitioner to obtain a good partitioning of the tasks. A limitation of this method appears when a data is shared by 3 (or more) tasks. Consider for example that some input data D_i is required by tasks T_a, T_b and T_c. In the modeled graph, this shared data leads to three edges (T_a, T_b), (T_a, T_c) and (T_b, T_c), leading METIS to count three times its weight. It is thus more reasonable to use a hypergraph to model data reuse, as proposed in [6]. A hyperedge is created for each data D which includes all the vertices corresponding to tasks using D as input. In the previous example, D_i is modeled with a single hyperedge $\{T_a, T_b, T_c\}$. We can then use a hypergraph partitioner, namely hMETIS [18] to split the tasks into subsets of similar size with few hyperedges between subsets, that is, few shared data. We mostly used default parameters when calling hMETIS [18], except for the default imbalance between partition (UBfactor), which is set to 1 in order to have almost perfectly balanced partitions, and V-cycle, set to 2, as it is advised by the authors if time is an issue (which is the case). The number of bisections (Nruns) is set to 20 as it is advised by the authors.

To ensure good temporal data locality within each subset of tasks assigned to a single GPU, we decided to use the *Ready* strategy of DMDAR (see Section IV-A) to refine the schedule produced by hMETIS. Additionally, even if the partition produced by METIS is well-balanced in terms of number of tasks, data transfers make some GPUs processing tasks faster than others, leading to load imbalance. Thereby, we also implement dynamic load balancing using task stealing: when a GPU has terminated its allocated tasks and some other GPU still has work to do, the idle GPU steals half of the remaining tasks from the GPU with the most unprocessed tasks

[1] https://files.inria.fr/starpu/testing/master/doc/html/Scheduling.html#DMTaskSchedulingPolicy

(starting at the tail of the list). We call this hMETIS+R.

Algorithm 3 hMETIS heuristic with ready (hMETIS+R)

1: For $j = 1, \ldots, m$, $h_j \leftarrow \{T_i,$ s.t. $D_j \in \mathcal{D}(T_i)\}$
2: Build hypergraph $H = (\{T_1, \ldots T_n\}, \{h_1, \ldots h_m\})$
3: Apply hMETIS on H to produce a task partition $P_1, \ldots, P_K$
4: Allocate tasks of P_k on GPU_k
5: If at some point GPU_k has no more tasks to process, steal half of the remaining tasks from the most loaded GPU and allocate them to GPU_k
6: Reorder tasks using Ready at runtime

C. Hierarchical Fair Packing adaptation to multi-GPU

We also consider here an algorithm recently proposed to order tasks sharing input data on a single GPU [14]. This algorithm is named HFP, for Hierarchical Fair Packing. It consists in gathering tasks sharing many common input data into packages, so that all inputs for a package fit in memory. These packages are then scheduled one after the other. The intuition is that once all the input data required for a package are loaded in memory, all tasks in that package can be processed without additional data movement.

HFP starts by considering that each task is a package of its own, and then merges two packages with fewest tasks that share the most common input data. Packages are merged that way as long as they do not exceed the memory bound. In a second step, resulting packages are merged again in order to bind together packages with high data affinity, so they can be scheduled one after the other. Packages are stored as lists so that we do not modify the order of tasks within packages when merging them, hence keeping the good data locality inside packages.

We adapted HFP for the multi-GPU case as follows. When scheduling tasks for K GPUs, we merge packages until we reach K of them. It is unlikely that all these packages represent the same load (computed as the number of tasks if tasks have the same duration, or else the total duration of tasks for heterogeneous tasks). To achieve load-balancing, we first compute the average load L_{avg} of a GPU. We then move the last tasks of the package $P_{\max}$ with highest load to the package $P_{\min}$ with smallest load in order to balance the load without exceeding L_{avg}. This process is repeated until the load is L_{avg} on all GPUs. The additional tasks are placed at the end of a package, as we noticed that there is usually more slack for communication near the end of the computation.

The previous static process is unable to provide a completely accurate load-balancing, as it is hard to predict the duration of communications on a shared bus as well as their overlap with computations. Thereby, we also implement for HFP the dynamic load balancing strategy using task-stealing introduced for hMETIS+R (see Section IV-B).

Finally, HFP uses the *Ready* reordering strategy from DMDAR to favor tasks with better data availability. The resulting strategy is called mHFP (for multi-GPU extension of HFP).

Algorithm 4 multi-GPU Hierarchical Fair Packing heuristic

1: Use HFP [14] to create K packages $P_1, \ldots, P_k$
2: $L_{avg} \leftarrow m/K$
3: **while** There exists P_i with $|P_i| > L_{avg}$ **do**
4: Let $P_{\max}$ be the largest package
5: Let $P_{\min}$ be the smallest package
6: Remove $\min(|P_{\max}| - L_{avg}, L_{avg} - |P_{\min}|)$ tasks from the tail of $P_{\max}$ and append them to $P_{\min}$
7: Allocate tasks of P_k on GPU_k
8: If at some point GPU_k has no more tasks to process, steal half of the remaining tasks from the most loaded GPU and allocate them to GPU_k
9: Reorder tasks using Ready at runtime

D. Data-Aware Reactive Task Scheduling

The previous strategies that partition the tasks in order to maximize data locality, namely hMETIS+R and mHFP, are static algorithms: they require a preliminary phase where the partition of tasks is computed, and whose complexity might be prohibitive for large number of tasks or data. We propose here a dynamic strategy, called DARTS (for Data-Aware Reactive Task Scheduling), adapted from previous algorithms specifically designed for linear algebra operations, such as outer products and matrix products [19]. The main idea of these algorithms is to perform as many tasks as possible with the data at hand. When a new data is loaded on a processor we allocate to this processor all the tasks that depend on the new data and on data previously loaded on this processor. New data are chosen at random to make sure different processors have little chance to compete on the same tasks.

The main idea of our new algorithm, detailed in Algorithm 5, is to first consider data movement before task allocation. Whenever some GPU_k requests some new task, we first look in the set $dataNotInMem_k$ (which initially contains all data) for the data D that, if moved into the memory of GPU_k, would maximize the number of new "free" tasks, i.e., tasks that can be allocated and processed on GPU_k without any additional data movement. Once such a data is found, all these free tasks are allocated on GPU_k. More precisely, they are put in a local data-structure ($plannedTasks_k$) where tasks are popped one after the other upon request. The process is started again when $plannedTasks_k$ is empty. It may happen that we do not find any data that enables some free task. It occurs for example at the very beginning of the computation when all tasks depend on two or more data: at least two data must be loaded in order to produce some free task. In this case, some random unprocessed task is allocated to GPU_k. On the contrary, when there exists several candidate data which may produce the maximum number of free tasks, we select a data among the candidates that is useful for the highest number of tasks (free or not). When a tie occurs (either in selecting a task or a data), we randomly pick some elements. This is important to make sure that different GPUs have little chance to load the same data and compete for the same tasks.

Algorithm 5 DARTS on GPU_k

When GPU_k requests a new task
if $plannedTasks_k \neq \emptyset$ **then**
  Return $pop(plannedTasks_k)$
else
  for each data $D \in dataNotInMem_k$ **do**
    Compute $n(D)$, the number of tasks that depend only on D and some data already loaded in memory
  Let $n_{\max}$ be the maximum of $n(D)$ for $D \in dataNotInMem_k$
  if $n_{\max} > 0$ **then**
    $Candidates \leftarrow$ set of data D with $n(D) = n_{\max}$
    Select $D_{opt} \in Candidates$ such that the number of unprocessed tasks depending on D_{opt} is maximum (break ties randomly)
    $plannedTasks_k \leftarrow$ set of unprocessed tasks depending only on D_{opt} and on other data already in memory
    $T \leftarrow pop(plannedTasks_k)$, remove D_{opt} from $dataNotInMem_k$
  else
    Select a random unprocessed task T, remove the inputs of T from $dataNotInMem_k$
  Return T

In order to improve the performance of our dynamic scheduler, we also designed a custom eviction policy: since we plan ahead which tasks will be allocated to a GPU (through the use of $plannedTasks$), we can take this information into account when we have to remove some data from the memory. This strategy is named Least Used in the Future (LUF) and is detailed in Algorithm 6. We consider all data currently in memory and check if they are used as input for a future task. There are two types of such future tasks: tasks in $plannedTasks_k$ that have been reserved for a later allocation on the GPU as mentioned above, but also tasks in $taskBuffer_k$, which are the tasks that have been popped from $plannedTasks_k$ for execution on GPU_k, and thus whose GPU placement cannot be changed any more (the required data for these tasks may already have been prefetched). We first try to evict a data which is not useful for any task in $taskBuffer_k$, and which is an input of few tasks in $plannedTasks_k$. If this is not possible, we apply Belady's rule [15] on tasks already allocated: we select the input data whose next usage in $taskBuffer_k$ is the furthest in the future, which is known to minimize data movement. In practice, this last rule is rarely used as we usually succeed finding a data not useful for any task in $taskBuffer_k$.

V. Experimental Evaluation

We present below a subset of the experimental evaluation conducted to compare the strategies presented above.[2]

[2]The code used to reproducibly obtain the results of this paper is available at: https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/IPDPS2021

Algorithm 6 Eviction procedure LUF for DARTS on GPU_k

for each data D in the memory of GPU_k **do**
  $n_b(D) \leftarrow$ number of tasks using D in $taskBuffer_k$
  $n_p(D) \leftarrow$ number of tasks using D in $plannedTasks_k$
if the minimum value of $n_b(D)$ on any data D is 0 **then**
  Select V such that $n_b(V) = 0$ and $n_p(V)$ is minimum
else
  Select V the data whose next use in $taskBuffer_k$ is the furthest in the future
Remove tasks depending on V from $plannedTasks_k$
Evict V from memory, push it to $dataNotInMem_k$

A. Settings

All strategies mentioned above have been implemented in the StarPU runtime system [3]. We performed both real experiments on tesla V100 GPUs (using cuBLAS 10.2 GPU kernels with single precision and 960×960 matrix tiles), as well as simulations using the ability to run StarPU code over the SimGrid simulator [20] to test our strategies in various experimental conditions. The use of simulation is motivated both by the fidelity of the simulated results as well as the saving of compute time and energy consumption, and the possibility to ignore the cost of the schedulers in simulation. We have most often limited the GPU memory to 500 MB in order to better distinguish the performance of different strategies even on small datasets.

Our main application scenario consists of tasks from a 2D matrix multiplication: the matrix product $C = A \times B$ is decomposed into tasks corresponding to the multiplication of one block-row of A with one block-column of B. Input data is thus the rows of A and columns of B; tasks are submitted row per row. We also use the following variations or extensions:

- The same application with a randomized task order.
- 3D matrix multiplication: the product is now decomposed into products of blocks of A and B. We do not here consider the final summation to concentrate on the computationally-intensive tasks without dependencies.
- Tasks coming from the Cholesky decomposition [21]: we remove dependencies among these tasks to deal only with independent tasks. This set of tasks exhibits some regularity, but is more complex than the 2D or 3D matrix multiplication.
- Sparse 2D matrix multiplication: we remove 98% of the tasks from the 2D matrix multiplication scenario above. This application scenario has a much larger communication-to-computation ratio, typical from sparse computations.

We use the four scheduling heuristics presented above, together with the baseline EAGER scheduler that lets GPUs pick up tasks on demand from a shared queue that contains the tasks in the natural order (i.e. row major for matrix multiplications). hMETIS+R is not used in the single GPU case. All the schedulers use the LRU's eviction policy except for DARTS+LUF.

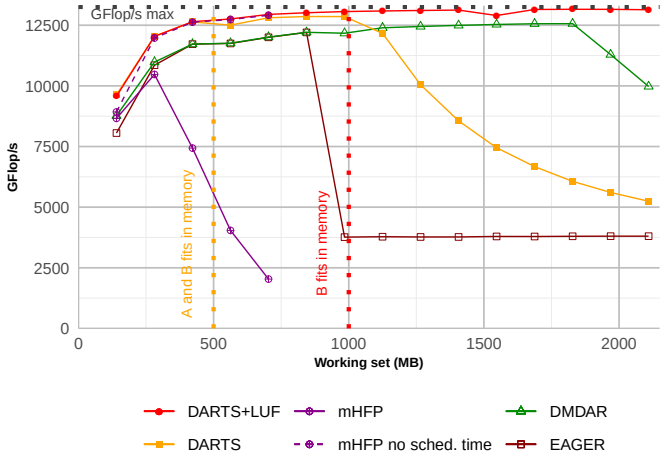

Figure 3: Performance on the 2D matrix multiplication in real with 1 Tesla V100 GPU.

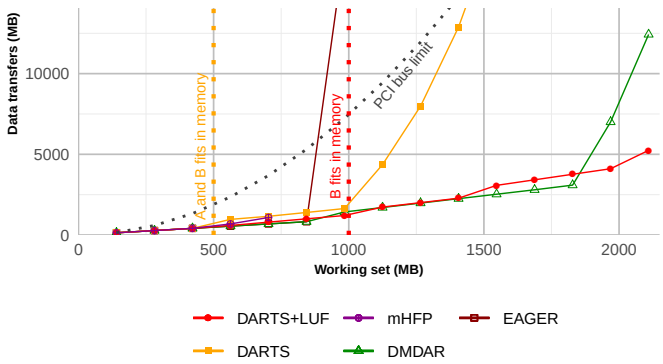

Figure 4: Amount of data transfers on the 2D matrix multiplication in real with 1 Tesla V100 GPU.

We measure the obtained performance as the throughput of elementary computational operations performed per time unit (in GFlop/s, thus the higher, the better), as well as the total volume of data transferred between CPU and GPUs (which we try to minimize). When measuring GFlop/s, the cost of computing the schedule is considered unless specified otherwise. Each result is the average of the performance obtained over 10 iterations. For most of the results, the deviance is less than 2%, thus, we do not show error bars in the following graphs. We plot the performance obtained with various problem sizes, the number of tasks ranging from 5×5 to 300×300 (which corresponds to a working set size of 140 MB to 8 400 MB) for the 2D matrix and up to 50 000 MB for the 3D matrix, in order to test all strategies on various memory conditions.

B. Results on a 2D matrix multiplication with a single GPU

Figures 3 and 4 show the results obtained by the various algorithms on one GPU.

a) General overview: On Figure 3 the dotted horizontal black line represents the maximum performance (13 253 GFlop/s) that the GPU can achieve when processing elementary matrix products (without I/Os) and is thus our asymptotic goal. The red dotted vertical line denotes the situation when the GPU memory (500 MB) can fit exactly only one of the two input matrices, and the orange line denotes the situation when it can accommodate both input matrices. Figure 4 shows the amount of data transfers. There, the black dotted curve represents the maximum number of transfers that can be done during the minimum time for computation (given by the bound on the throughput), thus the hard limitation induced by the PCI bus bandwidth: a strategy exceeding this amount necessarily requires more time for the data transfers than the optimal time for computation.

b) EAGER's results: The EAGER heuristic switches to a pathological behavior at the red line. We can both see the throughput plummeting (Figure 3) and the data transfers increasing (Figure 4) at the same working set size. EAGER tends to process tasks along the rows of C. This allows to reuse the same block-row of matrix A for tasks that compute tiles of the same row of C, but requires reloading the whole matrix B for each new block-row of A when the memory is constrained, which is a well-known pathological case of the LRU eviction policy. This explains why EAGER's performance drops at the red dotted line.

c) mHFP's results: We show two variants of mHFP for a few working set sizes on Figure 3. The dashed line represents the performance when we ignore the scheduling time, that is, when excluding the first phase in which the static task mapping is computed. We notice that it achieves very good performance. The continuous line represents the performance obtained while taking into account the scheduling time (like we do for every other heuristics). Unfortunately, the scheduling time of mHFP is very long for large working sets (1 minute for a 1 300 MB working set) and rapidly grows. Thus the overhead induced by the scheduling time overcomes it benefits.

d) DMDAR's results: DMDAR does not suffer from the pathological case affecting EAGER because its *Ready* strategy allows it to rather process tasks that need the block-column of B already in memory instead of reloading the whole B matrix. DMDAR's data transfers however start to rise for the last two working set sizes as we can see on Figure 4. That corresponds to the performance drop of the last two points of Figure 3. This is due to a conflict between data prefetching and eviction. Indeed, once the GPU is filled with data, it is not clear for DMDAR whether some data should be evicted in order to perform more prefetches. It will thus rather stop prefetching data as long as all the data currently in the GPU will be useful for the subsequent tasks to be executed there. Also, when some data is actually evicted from a GPU, DMDAR does not reconsider the task mapping according to the new set of data loaded on the GPU. The basic problem of DMDAR here is that it does not have a global view of the whole set of data and tasks, and thus cannot make a balance between prefetching and eviction.

e) DARTS' results: We can see that on the first seven points of Figure 3, DARTS and DARTS+LUF achieve near perfect performance. Indeed, loading a single data that enables

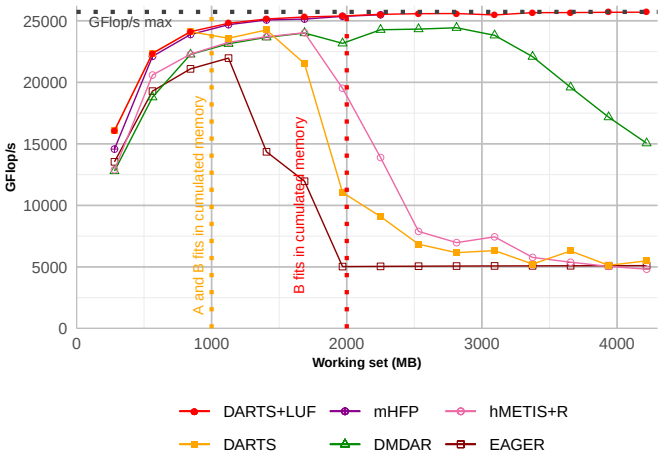

Figure 5: Performance on the 2D matrix multiplication in simulation with the performance models of 2 Tesla V100 GPUs.

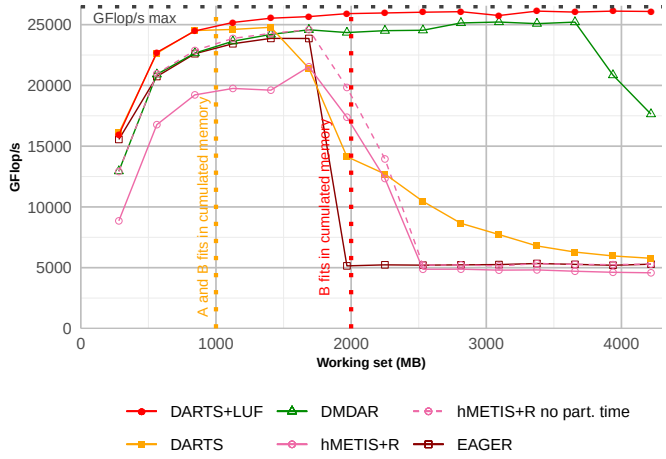

Figure 6: Performance on the 2D matrix multiplication in real with 2 Tesla V100 GPUs.

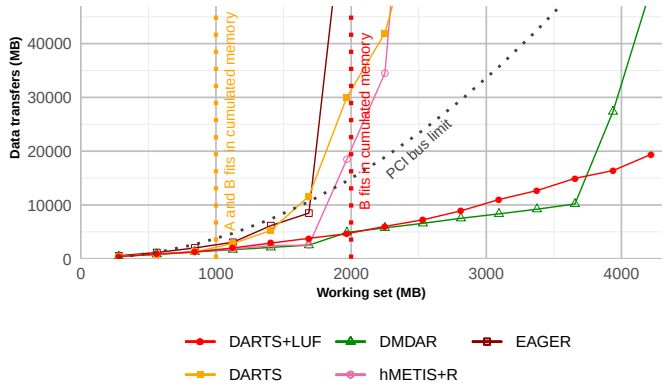

Figure 7: Amount of data transfers on the 2D matrix multiplication in real with 2 Tesla V100 GPUs.

multiple tasks minimizes data transfers and increases overlap between transfers and computations. However, after the red-dotted line, when the memory is constrained, DARTS has to load and evict data following LRU's policy. So the tasks in $taskBuffer$ that are supposed to be ready for computation, have to load more data. Indeed, the previous tasks caused evictions, and the data evicted might be needed by the tasks in $taskBuffer$. This causes a domino effect where each new task requires a new data load. On the contrary DARTS+LUF achieves on average 8.5% more GFlop/s than DMDAR. When an eviction is needed, it avoids as much as possible evicting a data that is used by the few tasks already planned for computation. This allows us to avoid the pathological case of DARTS, and to achieve a better balance between prefetching and eviction since DARTS+LUF maintains in $plannedTasks$ and $taskBuffer$ an accurate overview of the tasks to be computed, even when eviction removes some data from a GPU. It thus eventually achieves almost optimal performance.

C. Results on a 2D matrix multiplication with 2 or 4 GPUs

We now move to the multi GPUs case. Figures 5 shows the results obtained using simulation (for 2 GPUs), thus not taking into account the scheduling cost of all heuristics and the partition costs of hMETIS into account, while Figures 6 and 8 show the results obtained with real executions (for 2 or 4 GPUs). In this latter figures, we remove mHFP whose scheduling time is prohibitively large and added two version of hMETIS+R, one with partitioning time and one without (hMETIS+R no part. time) to show its impact on performance. Now the vertical lines depict the thresholds when one or both input matrices fit in the *cumulated* memory, that is, can be distributed over the memory of all GPUs.

a) EAGER's, hMETIS+R's and DARTS' results: Similarly to the single-GPU case, we observe on Figures 5 and 6 that EAGER, hMETIS+R and DARTS show lower performance under memory constraint. hMETIS+R gives a partitioning based on the data-sharing graph. In constrained situations, the lack of task ordering inside a partition, does not allow for good data reuse. The *Ready* heuristic can only reorder a limited number of tasks ahead of the computation and cannot improve performance by a significant margin. EAGER and DARTS both suffer from the same pathological case induced by the LRU strategy. By observing the two curves of hMETIS+R, we notice that the partitioning time of hMETIS+R has a significant impact on performance, and that this impact increases with the number of GPUs.

b) mHFP's results in simulation: As we can see on Figure 5, mHFP achieves very good performance when the scheduling time is not taken in consideration, showing that mHFP's strategy of load-balancing and task stealing achieves good results in theory. However, for mHFP, the scheduling time largely increases with the working set size. For a working set of 2000 MB for instance, mHFP takes more than 8 minutes. As was seen on Figure 3, the results of mHFP when accounting the scheduling time would thus be very poor. Thus for the following graph, we do not show mHFP on the plots.

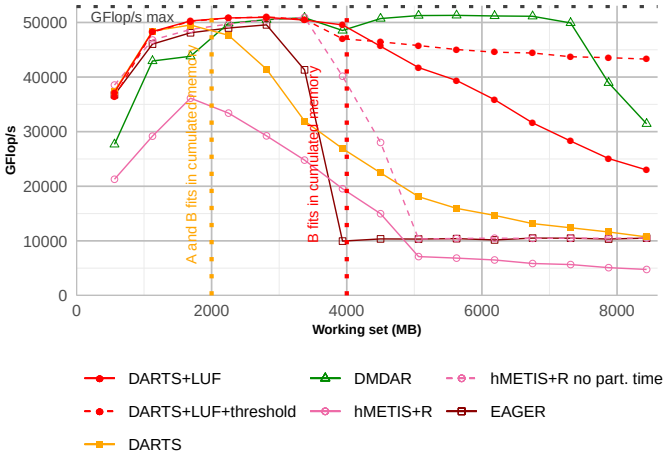

Figure 8: Performance on the 2D matrix multiplication in real with 4 Tesla V100 GPUs.

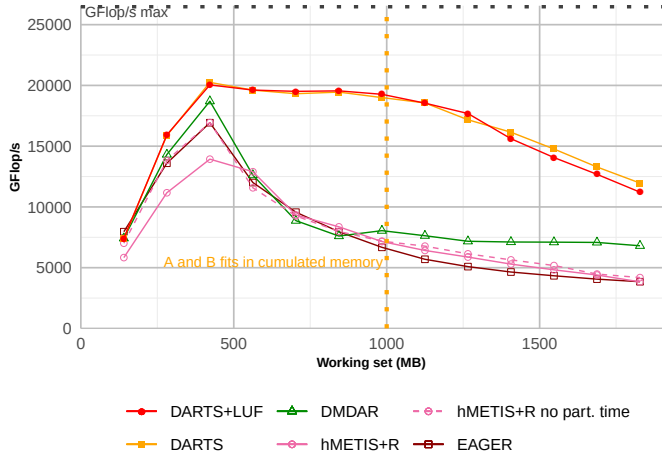

Figure 9: Performance on the 2D matrix multiplication with randomized task order in real with 2 Tesla V100 GPUs.

c) DMDAR's results: The DMDAR results on two GPUs are also very similar to the single-GPU case. DMDAR achieves a good load balance between the two GPUs and favors locality, but at some point it cannot properly manage both prefetching and LRU eviction like in the single-GPU case.

d) DARTS+LUF's results: DARTS+LUF gets performance close to ideal, with an 9.4% improvement over DMDAR for two GPUs, while maintaining a very low complexity. In multi-GPU, DARTS assigns to each GPU its own set of data $dataNotInMem_k$ to pick from. However all GPUs share the same set of tasks. Once a GPU is allocated a task, it is removed from the common set of available tasks. Thus our scheduler will naturally assign to the other GPUs data from a line or row that has not been used for tasks yet. This will evenly distribute tasks among GPUs and mostly separate data usage between GPUs. LUF's eviction policy allows us to keep the expected data loading order by evicting data that will be used the least for future tasks. Thus the scheduling can still be effective despite the memory constraint. It is also important to note that we observe in Figure 7 that DARTS+LUF has more data transfers than DMDAR between 2500 and 3500 MB. However, our throughput is always higher. This confirms that the overlap between calculations and transfers is effective. Indeed, separating data loads between several executions of tasks from $taskBuffer$ induces a better distribution of transfers. On the contrary DMDAR tends to load a large number of data at once for the computation of a new row of tasks.

e) Trends with more GPUs: Using 4 GPUs (as in Figure 8) mainly impacts the performance of DARTS: as we use larger task sets, the scheduling time required to find the optimal data to load begins to degrade the global performance of the strategy. To reduce the impact of scheduling time we have added a threshold on the number of data we can pick from when filling $plannedTasks$ for working sets larger than 3500 MB only (in line 4 of Algorithm 5). This reduces the quality of the scheduling for these working set sizes, but allows us to partially compensate for the performance drop (as we can see on the plots with the red dashed line), and to surpass DMDAR on the last two points of both plots. However, it is difficult to come up with an optimal threshold that limits scheduling time without impacting too much the schedule quality.

D. Result on a 2D matrix multiplication with a randomized task order with 2 GPUs

In order to test our heuristics in more irregular cases, we randomize the natural task order. It will also highlight the link between performance and tasks' submission order. Figure 9 shows that EAGER, DMDAR and hMETIS+R are highly impacted by the randomized order of submission as soon as the memory does not allow loading both input matrices. This shows that DMDAR actually relies on the natural order of tasks to get good performance in previous graphs. DARTS manages to maintain high throughput until the memory size is inferior to one input matrix size. On this graph DARTS+LUF achieves 75% more GFlop/s than DMDAR on average on all points.

E. Result on a 3D matrix multiplication with 4 GPUs

Figure 10 shows the results on a 3D matrix multiplication in simulation with 4 GPUs. We add here an additional variant for DARTS: "DARTS+LUF-3inputs". The interest of this variant comes into play when no data allows to compute a task without additional loads, i.e., in the "else" case on Line 13 of Algorithm 5. Instead of loading a random data, we first look for a data which enables as many tasks to be processed with a single additional data load as possible. Namely, we look for a data D such that the number of tasks depending on D, on another unloaded data D' and on some data already in memory is maximal. If we find such data D, we return any such task T, otherwise we return a random task. We observe on Figure 10 that this variant leads to a better schedule. It allows to reach a throughput about 61% larger than the one of

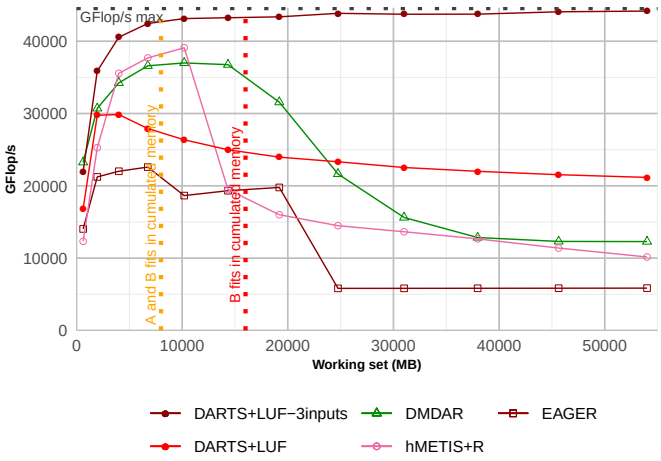

Figure 10: Performance on the 3D matrix multiplication in simulation with the performance models of 4 Tesla V100 GPUs.

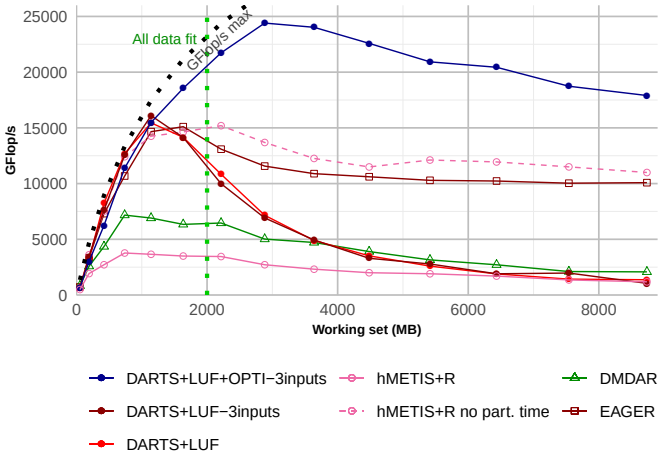

Figure 11: Performance on tasks coming from the Cholesky decomposition in real with 4 Tesla V100 GPUs.

DMDAR. It is important to note that from the second working set size, DARTS+LUF-3inputs is better than its competitors, which shows that even without memory limitation, the order of processing of our variant allows for a better overlap of tasks and data transfers.

F. Result on tasks coming from the Cholesky decomposition with 4 GPUs

Figure 11 shows the results on tasks coming from the Cholesky decomposition with 4 GPUs in real. Here, the green vertical line marks the working set size where all of input data fit into memory. We notice that DARTS is unable to achieve good performance, even with the *3inputs* variant. This is explained by the huge number of tasks and the resulting scheduling time. We thus enhance DARTS with the additional *OPTI* strategy to reduce scheduling time: instead of looking for the data that enables the most tasks, we stop the search as

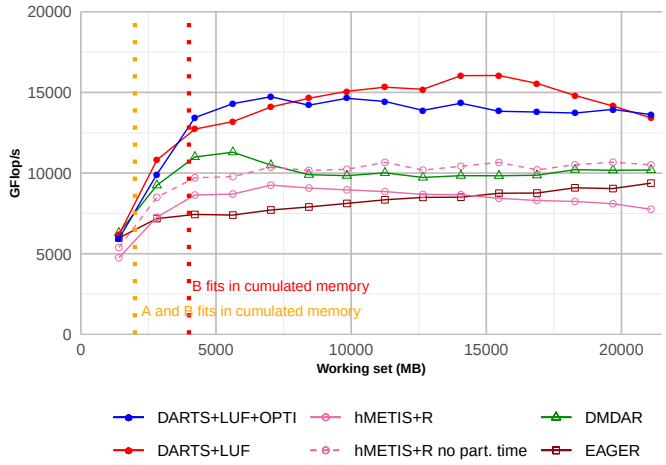

Figure 12: Performance on the sparse 2D matrix multiplication in real with 4 Tesla V100 GPUs.

soon as we find a data allowing to compute at least one task. This allows to maintain performance close to the optimal up to a 3000 MB working set. DARTS+LUF+OPTI-3inputs gets on average 49% more GFlop/s compared to hMETIS+R no part. time. Note that DMDAR also suffers from a large scheduling time induced by looking at all the tasks in order to choose the one allowing to avoid data loads. As a conclusion, DARTS can easily be extended to scenarios with more than 2 inputs per tasks, using the *3inputs* variants, as well as scenario with very large number of tasks, using *OPTI*.

G. Results on a sparse 2D matrix multiplication with 4 GPUs

Figure 12 shows the results on the sparse 2D matrix multiplication scenario, in which much less tasks can be computed with the same number of data. DARTS manages to navigate between sparse tasks without generating too many transfers, which is not the case for other schedulers. On this application we observe that DARTS+LUF obtains 40% more GFlop/s than DMDAR. As the total number of tasks is smaller than before, the *OPTI* variant is not needed, but we also see that it does not negatively impact the performance. Figure 13 shows the same application but without memory limitation. In this case, DARTS+OPTI obtains the best performance. This shows the ability of DARTS to produce a processing order that best distributes transfers over time. We also note that hMETIS suffers from an important partitioning cost; this largely decreases its performance that would otherwise be only slightly lower than DARTS.

VI. Conclusion and Future Work

Limiting data movements is crucial for performance in modern computing platforms, and especially for machines equipped with several GPUs sharing the same communication bus with the main memory. We have proposed several alternatives to schedule tasks sharing input data on such multi-GPU platforms, and implemented them over the StarPU runtime. We proposed a new strategy, named DARTS, which

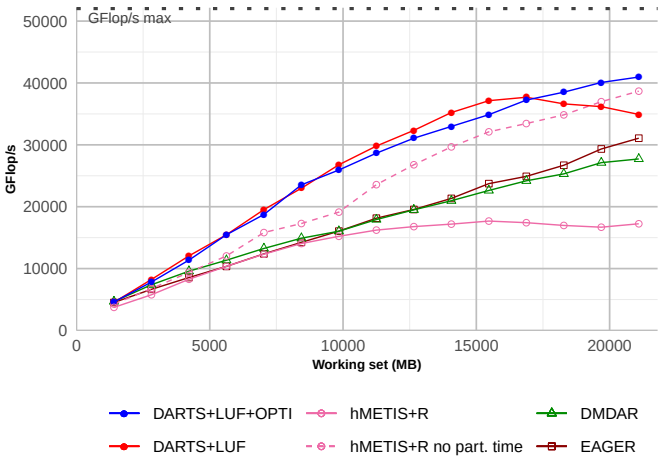

Figure 13: Performance on the sparse 2D matrix multiplication without memory limitation (32GB by GPU) in real with 4 Tesla V100 GPUs.

achieves very good performance both when the memory is not limited and when it is a scarce resource, and for applications with different data access patterns. We have shown that it can easily be extended to applications with many data per task and can deal with a large number of tasks using a dedicated optimization. However, for now it is only able to manage a single node with several GPUs and does not consider tasks with dependencies, which leaves us with several exciting future directions. Firstly, we would like to improve the computational complexity of DARTS to make it able to cope with a very large number of tasks, without sacrificing too much on the schedule quality. Secondly, we would like to adapt our model and algorithms to take inter-GPU communications into account, such as the one proposed by NVidia NVLinks, which enable fast data movement between pairs of GPUs without involving the CPU. Moving data from a nearby GPU is indeed usually faster than loading it from the main memory. In the long run, our objective is to consider tasks with dependencies and to mix data-locality objectives with other constraints such as task affinity with processing units, and task priorities in the graph of tasks.

Acknowledgments

This work was supported by the SOLHARIS project (ANR-19-CE46-0009) which is operated by the French National Research Agency (ANR).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 557–568.

[2] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, 2011.

[4] A. Giersch, Y. Robert, and F. Vivien, "Scheduling tasks sharing files on heterogeneous clusters," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*, ser. Lecture Notes in Computer Science. Springer, 2003, pp. 657–660.

[5] H. Senger, F. A. Silva, and W. M. Nascimento, "Hierarchical scheduling of independent tasks with shared files," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, 2006.

[6] K. Kaya and C. Aykanat, "Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave environments," *IEEE Trans. Parallel Distributed Syst.*, vol. 17, no. 8, 2006.

[7] J.-W. Hong and H. Kung, "I/O complexity: The red-blue pebble game," in *STOC'81: Proceedings of the 13th ACM symposium on Theory of Computing*. ACM Press, 1981, pp. 326–333.

[8] G. Kwasniewski, M. Kabic, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019*. ACM, 2019.

[9] T. M. Smith, B. Lowery, J. Langou, and R. A. van de Geijn, "A tight i/o lower bound for matrix multiplication," 2019, available at https://arxiv.org/abs/1702.02017.

[10] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: A quantitative limit study," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[11] J. V. Ferreira Lima, T. Gautier, V. Danjean, B. Raffin, and N. Maillard, "Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures," *Parallel Computing*, vol. 44, pp. 37–52, Mar. 2015.

[12] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," *Theory Comput. Syst.*, vol. 35, no. 3, pp. 321–347, 2002.

[13] K. Kaya, B. Uçar, and C. Aykanat, "Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories," *J. Parallel Distributed Comput.*, vol. 67, no. 3, 2007.

[14] M. Gonthier, L. Marchal, and S. Thibault, "Locality-Aware Scheduling of Independent Tasks for Runtime Systems," in *COLOC - 5th workshop on data locality - 27th International European Conference on Parallel and Distributed Computing*, Lisbon, Portugal, Aug. 2021, pp. 1–12.

[15] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, 1966.

[16] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *16th International Conference on Parallel and Distributed Systems*, Shangai, China, Dec. 2010.

[17] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[18] ——, "hMETIS 1.5 : A hypergraph partitioning package," Available at http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download.

[19] O. Beaumont and L. Marchal, "Analysis of dynamic scheduling strategies for matrix multiplication on heterogeneous platforms," in *The 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*. ACM, 2014, pp. 141–152.

[20] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, Jun. 2014.

[21] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, J. Roman, S. Thibault, and S. Tomov, "Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators." in *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Jul. 2010.