Locality-Aware Scheduling of Independant Tasks for Runtime Systems

Maxime Gonthier, Loris Marchal, Samuel Thibault

Locality-Aware Scheduling of Independent Tasks for Runtime Systems (Extended Version)

Maxime Gonthier, Loris Marchal , Samuel Thibault

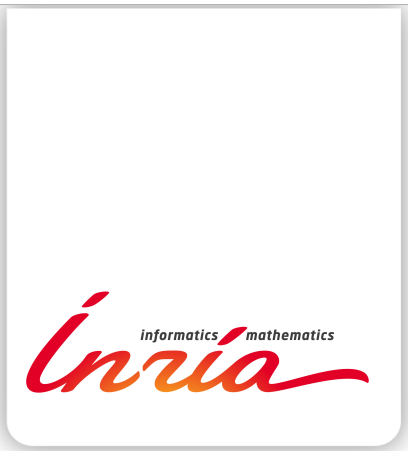
RESEARCH REPORT

N° 9394

February 2021

Project-Teams ROMA and STORM

ISSN 0249-6399 ISRN INRIA/RR--9394--FR+ENG



Locality-Aware Scheduling of Independent Tasks for Runtime Systems (Extended Version)

Maxime Gonthier *, Loris Marchal *, Samuel Thibault†

Project-Teams ROMA and STORM

Research Report n° 9394 — version 4 — initial version February 2021 — revised version March 2021 — 24 pages

Abstract: A now-classical way of meeting the increasing demand for computing speed by HPC applications is the use of GPUs and/or other accelerators. Such accelerators have their own memory, which is usually quite limited, and are connected to the main memory through a bus with bounded bandwidth. Thus, a particular care should be devoted to data locality in order to avoid unnecessary data movements. 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 order tasks and prefetch their input data in the GPU memory (after possibly evicting some previously-loaded data), while aiming at minimizing data movements, so as to reduce the total processing time. In this paper, we focus on how to schedule tasks that share some of their input data (but are otherwise independent) on a GPU. We provide a formal model of the problem, exhibit an optimal eviction strategy, and show that ordering tasks to minimize data movement is NP-complete. We review and adapt existing ordering strategies to this problem, and propose a new one based on task aggregation. These strategies have been implemented in the StarPU runtime system, which allows to test them on a variety of linear algebra problems. Our experiments demonstrate that using our new strategy together with the optimal eviction policy reduces the amount of data movement as well as the total processing time.

Key-words: Memory-aware scheduling, Eviction policy, Tasks sharing data, Runtime systems

* LIP, CNRS, ENS de Lyon, Inria & Université Claude-Bernard Lyon 1
† LaBRI, Université de Bordeaux, CNRS, Inria Bordeaux – Sud-Ouest

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Ordonnancement de tâches indépendantes pour support d'exécution utilisant la localité des données (Version Etendu)

Résumé : Une manière désormais classique de répondre à la demande croissante de puissance de calcul par les applications HPC est l'utilisation de GPU et autres accélérateurs. Ces accélérateurs ont leurs propre mémoire, qui est généralement assez limitée, et sont connectés à la mémoire principale via un bus dont la bande passante est bornée. Ainsi, une attention particulière doit être portée à la localité des données afin d'éviter des mouvements de données inutiles. Les ordonnanceurs des supports d'exécution à base de tâches sont un moyen pratique et efficace d'utiliser de telles plateformes hétérogènes. Lors du traitement d'une application, l'ordonnanceur a la connaissance de toutes les tâches disponibles, ainsi que leurs dépendances. Ainsi, il est capable d'ordonner les tâches et de pré-charger leurs données d'entrée dans la mémoire du GPU (après avoir éventuellement évincé certaines données précédemment chargées), tout minimisant les transferts de données, afin de réduire le temps d'exécution total. Dans ce papier, nous nous concentrons sur la façon de planifier des tâches qui partagent des données (mais sont par ailleurs indépendantes) sur un GPU. Nous fournissons un modèle formel du problème, nous présentons une stratégie d'éviction optimale et nous montrons qu'ordonner des tâches afin de minimiser les mouvement des données est un problème NP-complet. Nous adaptons des stratégies d'ordonnancement existantes à ce problème, et nous en proposons une nouvelle basé sur l'agrégation des tâches. Ces stratégies ont été implémentées sur le support d'exécution StarPU, ce qui permet de tester les ordonnanceurs sur une variété de problèmes d'algèbre linéaire. Nos expériences démontrent qu'en utilisant notre nouvelle stratégie, avec la politique d'éviction optimale, nous réduisons la quantité de transferts de données ainsi que le temps de traitement total.

Mots-clés : Ordonnancement sous contrainte mémoire, Politique d'éviction, Tâches partageant des données, Support d'exécution

1 Introduction

High-performance computing applications, such as physical simulations, molecular modeling or weather and climate forecasting, have an increasing demand in computer power to reach better accuracy. Recently, this demand has been met by extensively using GPUs, as they provide large additional performance for a relatively low energy budget. Programming the resulting heterogeneous architecture which merges regular CPUs with GPUs is a very complex task, as one needs to handle load balancing together with data movements and task affinity (tasks have strongly different speedups on GPUs). A deep trend which has emerged to cope with this new complexity is using task-based programming models and task-based runtimes such as PaRSEC [5] or StarPU [3]. These runtimes aim at scheduling scientific applications, described as directed acyclic graphs (DAGs) of tasks, onto distributed heterogeneous platforms, made of several nodes containing different computing cores.

Data movement is an important problem to consider when scheduling tasks on GPUs, as those have a limited memory as well as a limited bandwidth to read/write data from/to the main memory of the system. Thus, it is crucial to carefully order the tasks that have to be processed on GPUs so as to increase data reuse and minimize the amount of data that needs to be transferred. It is also important to schedule the transfers soon enough (prefetch) so that data transfers can be overlapped with computations and all tasks can start without delay. We focus in this paper on the problem of scheduling a set of tasks on one GPU with limited memory, where tasks share some of their input data but are otherwise independent. More precisely, we want to determine the order in which tasks must be processed as well as when their input must be loaded/evicted into/from memory. Our objective is to minimize the total amount of data transferred to the GPUs for the processing of all tasks. 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 subset of tasks, which are independent of each others and the size of this subset is usually quite large. This is in particular the case with linear algebra workflows, such as the matrix multiplication or Cholesky decomposition: except possibly at the very beginning or very end of the computation, a large set of tasks is available for scheduling. Thus, solving the optimization problem for the currently available tasks can lead to a large reduction in data transfers and hence a performance increase.

In this paper, we make the following contributions:

- We provide a formal model of the optimization problem, and prove the problem to be NP-complete. We derive an optimal eviction policy by adapting Belady's rule for cache management (Section 2).
- We review and adapt three heuristic algorithms from the literature for this problem, and propose a new one based on gathering tasks with similar data patterns into packages (Section 3).
- We implement all four heuristics into the StarPU runtime and study the performance (amount of data transfers and total processing time) obtained on various tasks sets coming from linear algebra operations (Section 4). 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 heterogeneous computing: 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 shared memory, and limited bandwidth for the communication between memory and disk.

The problem of fitting a working set in a given amount of memory [9] is a general concern that shows up at various levels such as the processor cache, the operating system page cache (most often solved with only a mere LRU heuristic), or tasks sharing files in a distributed context [13]. By some aspects, our work is for instance close to some studies in cache optimization, which consists in reordering requests to optimize cache reuse [10]. However, contrarily to these studies, we have here complete access to the whole set of tasks, not only a subset, and can completely re-order it. The MST approach [15] is probably closest to our work since it focuses on ordering independent tasks sharing data. They target multicore CPUs, which entails addressing both grouping and ordering tasks. In the experiments presented in this paper, we compare our heuristic to the MST approach.

2 Problem modeling and complexity

We consider the problem of scheduling independent tasks on a GPU with memory size M. 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 iff 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 consider that all data have the same size. The GPU is equipped with a memory of limited size, which may contain no more than M data simultaneously. During the processing of a task T_i, all its inputs $\mathcal{D}(T_i)$ must be in memory.

For the sake of simplicity, we here do not consider the data output 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.

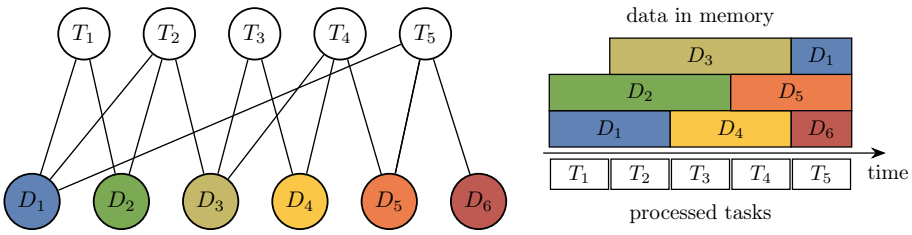


Figure 1: Example with 5 tasks and 6 data, with a memory holding at most $M = 3$ data. The graph of input data dependencies is shown on the left. The schedule on the right corresponds to processing the tasks in the natural order with the following eviction policy: $\mathcal{V}(1) = \mathcal{V}(2) = \emptyset$, $\mathcal{V}(3) = \{1\}$, $\mathcal{V}(4) = \{2\}$, $\mathcal{V}(5) = \{3, 4\}$. This results in 7 loads (only D_1 is loaded twice).

All tasks must be processed. Our goal is to determine in which order to process them, and when each data must be loaded or evicted, in order to minimize the amount of data movement. More formally, we denote by σ the order in which tasks are processed, and by $\mathcal{V}(t)$ the set of data to be evicted from the memory before the processing of task $T_{\sigma(t)}$. A schedule is made of m steps, each step being composed of the following three stages (in this order):

1. All data in $\mathcal{V}(t)$ are evicted (unloaded) from the memory;
2. The input data in $\mathcal{D}(T_{\sigma(t)})$ that are not yet in memory are loaded;
3. Task $T_{\sigma(t)}$ is processed.

An example is shown in Figure 1. In this model, input data are loaded in memory 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 we do not consider this in our model: we may simply book part of our memory for the pre-fetch mechanism.

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

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

Our memory limitation can then be expressed as $|L(t)| \leq M$ for each step $t = 1, \ldots, m$. Our objective is to minimize the amount of data movement, i.e., to minimize the number of *load* operations: we consider that data are not modified so no *store* operation occurs when evicting a data from the memory. Assuming that no input data used at step t is evicted right before the processing ($\mathcal{V}(t) \cap \mathcal{D}(T_{\sigma(t)}) = \emptyset$), the number of loads can be computed as follows:

$$\#Loads(\sigma, \mathcal{V}) = \sum_t \left| \mathcal{D}\left(T_{\sigma(t)}\right) \backslash L(t) \right|$$

There is no reason for a scheduling policy to evict some data from memory if there is still room for new input data. We call *thrifty scheduler* such a strategy, formalized by the following constraints: if $\mathcal{V}(t) \neq \emptyset$, then $|L(t)| = M$. For this class of schedulers, the number of loads can be computed more easily: as soon as the memory is full, the number of loads is equal to the number of evictions. That is, for the regular case when not all data fit in memory ($n > M$), we have:

$$\#Loads(\sigma, \mathcal{V}) = M + \sum_t |\mathcal{V}(t)|$$

Our optimization problem is stated below:

Definition 1 (MinLoadsForTasksSharingData)**.** *For a given set of tasks $\mathbb{T}$ sharing data in $\mathbb{D}$ according to $\mathcal{D}$, what is the task order σ and the eviction policy $\mathcal{V}$ that minimizes the number of loads $\#Loads$?*

A solution to this optimization problem consists in two parts: the order σ of the tasks and the eviction policy $\mathcal{V}$. An optimal eviction policy has been proposed for cache management policy, which corresponds to the same problem when tasks request a single data [14]: when the memory (or the cache) is full and new data must be loaded, an optimal policy consists in evicting the data whose next use is the furthest in the future. Hence, it requires to know the full series of future requests and is only usable in the offline case. This is the well-known Belady MIN replacement policy [4]. We prove in the following theorem that this rule can be extended to our problem, with tasks requiring multiple data.

Theorem 1. *We consider a task schedule σ for a* MinLoadsForTasksSharingData *problem. We denote by MIN the thrifty eviction policy that always evicts a data whose next use in σ is the latest (breaking ties arbitrarily). MIN reaches an optimal performance, i.e., for any eviction policy $\mathcal{V}$,*

$$\#Loads(\sigma, MIN) \leq \#Loads(\sigma, \mathcal{V}).$$

Proof. We consider a given task order σ. We transform our problem so that each task depends on a single data (or page). we replace a task T_i depending on data $\mathcal{D}(T_i) = \{D_1, \ldots, D_k\}$ by a series of $2k$ tasks: $T_i^{(1)}, T_i^{(2k)}$ such that $\mathcal{D}(T_i^{(j)}) = \mathcal{D}(T_i^{(j+k)}) = D_j$ for $j = 1, \ldots, k$. We denote by $\mathbb{T}'$ the modified set of tasks and by σ' the modified task order.

Let $\mathcal{V}$ be an optimal eviction policy for the original problem, i.e. for task set $\mathbb{T}$ and task order σ. We now transform it into an eviction policy for $\mathbb{T}'$ and σ' with the same number of loads and evictions. We group tasks by subsets of $2k$ tasks (as they were created above) and evict all data in $\mathcal{V}(t)$ before processing tasks $T^{(1)}_{\sigma(t)}, T^{(2k)}_{\sigma(t)}$ (and loading their missing inputs). We denote this strategy by $\mathcal{V}'$. Clearly, this is a valid strategy (we never exceed the memory if $\mathcal{V}$ did not on the original problem) and it has the same number of loads as $\mathcal{V}$:

$$\#Loads(\sigma, \mathcal{V}) = \#Loads(\sigma', \mathcal{V}').$$

Symmetrically, we consider an optimal eviction policy for the transformed problem ($\mathbb{T}'$ and σ') obtained with Belady's MIN replacement policy, denoted by MIN': whenever some data must be evicted, it selects the one whose next use is the furthest in the future. We now prove that it can be transformed into an eviction policy MIN for the original problem with the same performance (loads and evictions), and that MIN also follows Belady's rule. We consider a subset of $2k$ tasks $T^{(1)}_i, T^{(2k)}_i$ coming from the expansion of task T_i scheduled at time t ($\sigma(t) = T_i$) and the set of data V evicted by MIN' right before some task $T^{(j)}_i$. By property of MIN' and as the memory is large enough for the inputs of task T_i ($M \geq k$) no input data of some $T^{(j)}_i$ belongs to V: during the first k tasks, their next occurrence is the closest, and there is no eviction during the processing of the last k tasks. Thus, we can adapt MIN' for the original problem by setting $MIN(t) = V$. It is easy to verify that MIN reaches the same performance as MIN':

$$\#Loads(\sigma, MIN) = \#Loads(\sigma', MIN')$$

and that the data evicted at time t are (among the) ones whose next use is the furthest in the future.

As MIN' is known to be optimal for the transformed problem, we have $\#Loads(\sigma', MIN') \leq \#Loads(\sigma', \mathcal{V}')$ and we conclude that $\#Loads(\sigma, MIN) \leq \#Loads(\sigma, \mathcal{V})$, which proves that MIN is optimal on the original problem. □

For cache management, Belady's rule has little practical impact, as the stream of future requests is generally unknown; simple online policies such as LRU (Least Recently Used [9]) are generally used. However in our case, the full set of tasks is available at the beginning. Hence, we can greatly take advantage of this optimal offline eviction policy, as our experiments demonstrate below.

Thanks to the previous result, we can restrict our problem to finding the optimal task order σ. Unfortunately, this problem is NP-complete.

Theorem 2. *For a given set of tasks $\mathbb{T}$ sharing data in $\mathbb{D}$ according to $\mathcal{D}$ and a given integer B, finding a task order σ such that $\#Loads(\sigma, MIN) \leq B$ is NP-complete.*

Proof. We first check that the problem is in NP. Given a schedule σ (and an eviction policy $\mathcal{V}$, which might be deduced by MIN), it is easy to check in polynomial time that:

- The schedule is valid, that is, no more than M data are loaded in memory at any time step;
- The number of loads is not greater than the prescribed bound B.

The NP-completeness proof consists in a reduction from the cutwidth minimization problem (or CMP), proven NP-complete by Gavril in 1977 [12]. We denote by I_{CMP} an instance of CMP composed of a graph. The question is to decide whether there exists a linear arrangement of the vertices such that the cutwidth is at most K. A linear arrangement α is a simple order of the vertices. The cutwidth $CUT_\alpha(v)$ of a vertex v under the linear arrangement α is the number

of edges that connect vertices ordered before and after v in α, that is, the number of edges $(u, w) \in E$, such as $\alpha(u) < \alpha(v) < \alpha(w)$. The total cutwidth of G is the maximal cutwidth over all vertices : $CUT_\alpha(G) = \max_{v \in V} CUT_\alpha(v)$.

Given an instance I_{CMP} an instance of *CMP*, we create an instance $I_{MinLoads}$ of our problem as follows. For each vertex $v_i \in I_{CMP}$, we create a task T_i, and for each edge $e_k = (v_i, v_j)$, we create a data D_k such that D_k is a shared input of T_i and T_j. Then,

$$\mathcal{D}(T_i) = \{D_k, \text{ such that } e_k \text{ is adjacent to } v_i \text{ in } G\}.$$

Finally, we set $M = K$ and $B = |\mathbb{D}| = |E|$: we are looking for a solution where each data is loaded exactly once.

We now prove that if I_{CMP} has a solution, then $I_{MinLoads}$ has a solution. Let α be the linear arrangement solution of I_{CMP}. We consider the task order $\sigma = \alpha^{-1}$, and the optimal eviction policy *MIN*. We prove that

(i) A data is evicted only if it is not used anymore;

(ii) Each data is loaded exactly once.

Note that (ii) is a direct consequence of (i). We consider a step t when some data D_j is evicted and some task T_i is processed. We consider the set S of data in memory before starting step t together with the inputs of $T_{\sigma(t)}$ that are loaded in memory during step t. If D_j is evicted, this means that $|S| > M$ (*MIN* is a thrifty policy). We consider S', the subset of S containing the data that are as input for a later step $t' > t$. By construction of $I_{MinLoads}$, each data $D_k \in S'$ corresponds to an edge $e_k = (v_a, v_b)$ in G such that $\sigma^{-1}(u_a) = \alpha(v_a) < t$ (the data was loaded for a task T_a scheduled before t) and $\sigma^{-1}(v_b) = \alpha(v_b) > t$ (the data is used for a task T_b scheduled after t). Hence, it corresponds to an edge counted in the cutwidth $CUT_\alpha(v_i)$. Since this cutwidth is bounded by $K = M$, there are at most M data in S'. Thus, the evicted data D_j is not used later than t. Since all data are loaded exactly once, the number of loads is not larger than B.

We now prove that if $I_{MinLoads}$ has a solution, then I_{CMP} has a solution. Let σ the task order in the solution of $I_{MinLoads}$. We construct the solution of I_{CMP} such that $\alpha = \sigma^{-1}$. We now prove that its cutwidth is not larger than K. By construction, the cutwidth $CUT_\alpha(v_i)$ at some vertex v_i (corresponding to a task T_i scheduled at time t) is the number of data which are used both before t and after t. Given the constraint on the number of loads, each data is loaded once, so such a data must be in memory during the processing of T_i, and there are at most M such data. This proves that $CUT_\alpha(v_i) \leq M = B$. Hence α is a solution for I_{CMP}. □

3 Algorithms

We present here several heuristics to solve the MinLoadsForTasksSharingData optimization problem. Two of them are adapted from the literature (Reverse-Cuthill-McKee and Maximum Spanning Tree), one of them is the actual dynamic strategy from the StarPU runtime (DMDAR) and we finally propose a new strategy (HFP).

Reverse-Cuthill-McKee (RCM) We have seen above that our problem is close to the cutwidth minimization problem, known to be NP-complete. This motivates the use of the Cuthill–McKee algorithm, which concentrates on a close metric: the bandwidth of a graph. It permutes a sparse matrix into a band matrix so that all elements are close to the diagonal [8]. If the resulting bandwidth is k, it means that vertices sharing an edge are not more than k edges away. We apply this algorithm on the graph of tasks $G^T = (\mathbb{T}, E^T, w^T)$ where there is an edge

(T_i, T_j) if tasks T_i and T_j share some data, and where $w^T(T_i, T_j)$ is the number of such shared data. If the bandwidth of the graph is not larger than k, this means in our problem that any task T_i processed at time t has all its "neighbours" tasks (tasks sharing some data with T_i) processed in the time interval $[t-k; t+k]$. Hence, if k is low, this leads to a very good data locality. Reversing the obtained order is known to improve the performance of the Cuthill–McKee algorithm, which we also notice in our experiments. The adaption of the Reverse-Cuthill–McKee algorithm is described in Algorithm 1.

Algorithm 1 Reverse-Cuthill-McKee heuristic

Build the graph G^T where vertices are tasks and edges are common data between tasks, weighted by the number of such data
$\sigma \leftarrow [v]$ where v is the vertex of G^T with smallest weighted degree
$i \leftarrow 0$
while $|\sigma| < m$ **do**
  Let N be the set of vertices adjacent to $\sigma[i]$ in G^T not yet in σ
  Sort N by non-decreasing weighted degree
  Append N at the end of σ
  $i \leftarrow i + 1$
end while
Return σ in the reverse order

Differences CM and RCM We prove in the following theorem that both Cuthill-McKee (CM) and Reverse-Cuthill-McKee (RCM) algorithms reach the same amount of data movement. More generally, reversing a schedule does not change the number of reads or eviction.

Theorem 3. *For a given set of tasks $\mathbb{T}$ sharing data in $\mathbb{D}$ and a given task order σ : $\#Loads(\sigma, MIN) = \#Loads(\bar{\sigma}, MIN)$.*

Proof. Given σ, an order of computation for $\mathbb{T}$, we know that data are used in the following order: $\mathcal{D}(T_{\sigma(1)}), \ldots, \mathcal{D}(T_{\sigma(m)})$. Together with the knowledge of the *MIN* eviction policy, we can deduce the set of data that we need to load before computing task $T_{\sigma(t)}$, that we note S_t. It is the set of input data of T_i that were not in memory during the computation of the last task T_{i-1}, in other words: $S_t = \mathcal{D}(T_{\sigma(t)}) \backslash L(t-1)$. We denote by $\mathbb{S}$ the ordered list of data sets that we need to load before each task: $\mathbb{S} = [S_1, ..., S_m]$. Similarly, we build $\mathbb{V}$, the ordered list of data that we are evicted before each task= $\mathbb{V} = [\mathcal{V}(2), ..., \mathcal{V}(m), \mathcal{V}(m+1)]$. Note that we start at task 2 (no data is evicted before the first task) and we denote by $\mathcal{V}(m+1)$ the operation needed to completely empty the memory at the end of the execution. $\mathbb{S}$ and $\mathbb{V}$ totally describe the memory operations for an execution, and can be used to count the number of loads:

$$\#Loads(\sigma, MIN) = \#Loads_{ordered\_list}(\mathbb{S}, \mathbb{V}) = \sum_{S_i \in \mathbb{S}} |S_i| = \sum_{\mathcal{V}(i) \in \mathbb{V}} |\mathcal{V}(i)|$$

The last equality comes from the fact that each data is evict exactly as many times as it is loaded, thanks to the last eviction that totally frees the memory.

We consider the reversed order of σ: $\bar{\sigma}$, and similarly the reversed list of loads ($\bar{\mathbb{S}}$) and evictions ($\bar{\mathbb{V}}$). We consider $\mathbb{S}' = \bar{\mathbb{V}}$ and $\mathbb{V}' = \bar{\mathbb{S}}$ and notice that the pair $(\mathbb{S}', \mathbb{V}')$ describes correct lists of loading sets and eviction sets for $\bar{\sigma}$: this is what happens if we reverse the task order, and consider than each eviction for σ is transformed into a load, and each load for σ is transformed into an eviction. Hence, the total memory used by $(\mathbb{S}', \mathbb{V}')$ for $\bar{\sigma}$ is the same as the one used by

$(\mathbb{S}, \mathbb{V})$ for σ, and not larger than M. Because $(\mathbb{S}', \mathbb{V}')$ is a correct loading/eviction scheme, we have

$$\#Loads_{ordered\_list}(\mathbb{S}', \mathbb{V}') \leq \#Loads(\bar{\sigma}, MIN)$$

We also have

$$\begin{aligned} \#Loads_{ordered\_list}(\mathbb{S}', \mathbb{V}') &= \#Loads_{ordered\_list}(\bar{\mathbb{V}}, \bar{\mathbb{S}}) \\ &= \sum_{S_i \in \mathbb{S}} |S_i| \\ &= \#Loads_{ordered\_list}(\mathbb{S}, \mathbb{V}) = \#Loads(\sigma, MIN) \end{aligned}$$

Hence, we have $\#Loads(\bar{\sigma}, MIN) \leq \#Loads(\sigma, MIN)$ By reversing once again the schedule (as well as the list of loading sets and eviction set), we obtain similarly that $\#Loads(\sigma, MIN) \leq \#Loads(\bar{\sigma}, MIN)$, proving the equality. □

As we will see later (see Section 4, Figure 8), the performance reached by both variants are not similar. We observed that in practice, RCM is always better than CM. Even if the total number of load is the same, the distribution of loads in time is not equal: there is more overlap between data movements and computation in RCM than in CM, which allows RCM to reach better performance.

Maximum Spanning Tree (MST) As outlined in the related work, another heuristic has been proposed to order tasks sharing data to improve data locality [15]. The authors of this study propose to build a Maximum Spanning Tree in the graph G^T using Prim's algorithm [7] and to order the vertices according to their order of inclusion in the spanning tree. By selecting the incident edge with largest weight, we increase the data reuse between the current scheduled tasks and the next one to process. The adaption of the Maximum Spanning Tree algorithm is described in Algorithm 2

Algorithm 2 Maximum Spanning Tree heuristic

```
For each vertex v_i set Key_Value(v_i) to 0
Key_Value(v_0) ← 1
while |σ| ≠ m do
    Choose v_i ∈ T ∉ σ such that Key_Value(v_i) is maximum
    Add v_i at the end of the list σ
    For each couple (v_i, v_j), update Max_Path_Length(i, j)
    for each v_j adjacent to v_i∩ ∉ σ do
        if Key_Value(v_j) < Max_Path_Length(i, j) then
            Key_Value(v_j) ← Max_Path_Length(i, j)
        end if
    end for
end while
Return σ
```

Deque Model Data Aware Ready (DMDAR) DMDA or "Deque Model Data Aware" is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing units in the StarPU runtime. It takes data transfer time into account and schedules tasks where their completion times is expected to be minimal [2] (also called tmdp). We focus here on a variant,

DMDAR, which additionally favors tasks whose data has already been loaded into memory. In our context with a single processing unit, it is reduced to selecting the next task as the one that has the minimum number of inputs not already loaded in memory (see Algorithm 3). DMDAR is a dynamic scheduler that relies on the actual state of the memory, it thus depends on the eviction policy, which is the LRU policy.

Algorithm 3 DMDAR heuristic

Let T_1 be the first task in $\mathbb{T}$
Add T_1 to σ, $InMem \leftarrow \mathcal{D}(T_1)$
while $|\sigma| \neq m$ **do**
  Select a task $T_i \notin \sigma$ such that $|\mathcal{D}(T_i) \backslash InMem|$ is minimal
  Add T_i to σ
  $InMem \leftarrow InMem \cup \mathcal{D}(T_i)$
  if $|InMem| > M$ **then** evict data from $InMem$ following LRU's policy
end while
Return σ

Hierarchical Fair Packing (HFP) HFP builds packages (denoted $P_1, P_2, ...$) of tasks, which are stored as lists of tasks, forming a partition of $\mathbb{T}$. To do so, it gathers tasks that share the most input data. By extension, we denote by $\mathcal{D}(P_k)$ the set of inputs of all tasks in P_k. We aim at building the smallest number of packages so that for each package, $\mathcal{D}(P_k) \leq M$. The intuition is that once the data $\mathcal{D}(P_k)$ are loaded, all tasks in the package can be processed without any additional data movement. We have proven that building the minimum number of packages is NP-complete(see Theorem 4 and it's proof), hence we rely on a greedy heuristic to build them, described in Algorithm 4.

Theorem 4. *We consider a set of tasks $\mathbb{T}$ sharing data in $\mathbb{D}$. Partitioning tasks into at most L packages $P_1, \ldots, P_L$ such that $|\mathcal{D}(P_i)| \leq M$ for each package P_i is an NP-complete problem.*

Proof. Given a set of packages, it is easy to verify that they partition the task set and that the input size of each package is smaller than the M bound, hence the problem lies in NP.

We prove that the problem is NP-complete thanks to a reduction from the 3-partition problem: Given an integer B and $3n$ integer $a_1, a_2, \ldots, a_n$, such that $\sum_{i=1}^{3n} a_i = nB$ the problem is to decide whether we can partition $3n$ into n triplet whose sum of integers is B. We consider the restricted version of the problem where $\forall i \ M/4 < a_i < M/2$, which is still NP-complete [11]. In this variant, each subset of integers reaching B has exactly three elements.

We consider an instance I_{3P} of the 3-partition problem and build an instance I_{MinP} of the package minimization problem as follows. For each $a_i \in I_{3P}$, we create a task T_i and a_i input data $\mathcal{D}(T_i) = \{D_{i,1} \ldots, D_{i,a_i}\}$ (no input data is shared amond two tasks). We set the size limit of a package to $M = B$ and the maximum number of packages to $L = n$. Thus, in instance I_{MinP}, we try to solve the following question: Can we find at most n packages of input size at most M?

We know prove that if I_{3P} has a solution, then I_{MinP} has a solution. If I_{3P} has a solution, then we have n subsets of integers $S_1, S_2, \ldots, S_n$ which verify: $\forall i, |S_i| = M$. We group tasks in n packages $P_1, P_2, \ldots, P_n$ such that $P_j = \{T_i, a_i \in S_j\}$ Since, $L = n$, we have exactly L packages. The input size of each package is

$$|\mathcal{D}(P_j)| = \sum_{T_i \in P_j} |\mathcal{D}(T_i)| = \sum_{a_i \in S_j} a_i = B = M.$$

Hence, this is a solution for I_{MinP}.

We know prove that if I_{MinP} has a solution, then I_{3P} has a solution. If I_{MinP} has a solution then there are at most L packages whose input size is at most M: $\forall i |\mathcal{D}(P_i)| \leq M$. We know that $\sum_{i=1}^{n} |S_i| = nM$, so $\sum_{i=1}^{n} |\mathcal{D}(P_i)| = nM$. We therefore have $L = n$ packages which must satisfy the following conditions:

$$\begin{cases} \sum_{i=1}^{n} |\mathcal{D}(P_i)| = nM \\ \forall i \ |\mathcal{D}(P_i)| \leq M \end{cases}$$

Any package with input size smaller than M would require that another package has a size larger than M, which is not possible. Therefore, we have $|\mathcal{D}(P_i)| = M$ for each package P_i. We denote by S_j the set of a_i corresponding to tasks T_i in P_j. Hence, $\sum_{a_i \in S_j} a_i = M$ for all S_j. We assume that $M/4 < a_i < M/2$, hence each S_j counts exactly three a_is, and the S_j are a solution to instance I_{3P}. □

Since building packages in an optimal way is NP-complete, we concentrate on a greedy heuristic to build them, described in Algorithm 4. We start with packages containing a single task. Then we consider all packages with fewest tasks and try to merge each of them with another package with whom it shares the most input data. When it is not possible to merge packages without exceeding the M bound, we perform a second step where we gather packages in the same way but ignore the bound M on the input size. The intuition is to create meta-packages that express the data affinity between packages already built. Note that we do not modify the order of tasks within packages when merging them, hence keeping the good data locality inside packages. Eventually, the last remaining package after all merges is the list of tasks for the schedule.

Algorithm 4 Hierarchical Fair Packing heuristic

```
Let P_i ← [T_i] for i = 1...m and ℙ = {P_1, ..., P_m}
SizeLimit ← true, MaxSizeReached ← false,
while |ℙ| > 1 do
    while (MaxSizeReached = false or SizeLimit = false) and |ℙ| > 1 do
        MaxSizeReached ← true
        for all packages P_i with the smallest number of tasks do
            Find a package P_j such that |D(P_i) ∩ D(P_j)| is maximal
            if weight(P_i ∪ P_j) ≤ M or SizeLimit = false then
                Merge P_i and P_j: append P_j at then end of P_i and remove P_j from ℙ
                MaxSizeReached ← false
            end if
        end for
    end while
    SizeLimit ← false
end while
Return the only package in ℙ
```

Complexity of HFP Let's note $\Delta = \max_i |\mathcal{D}(T_i)|$ the maximal number of data for any task. In the best case we merge all the packages two by two at each iteration of the inner while loop. It result in $\log_2 m$ iterations. The number of data of a package is growing at each fusion, it size is at most $\Delta \times 2^i$. At iteration i we have $\frac{m}{2^i}$ packages and $\Delta \times 2^i$ data by package. To find the

package with which a package share the most data we must compute $(\frac{m}{2^i})^2$ on the $\Delta \times 2^i$ data of each package. So the cost at iteration i is:

$$(\frac{m}{2^i})^2 \times \Delta \times 2^i = \frac{m^2 \times \Delta}{2^i}$$

If we sum the i iterations we get:

$$\Delta \times m^2 + \frac{\Delta \times m^2}{2^1} + \frac{\Delta \times m^2}{2^2} + ... + \frac{\Delta \times m^2}{2^i} < O(\Delta \times m^2)$$

So the complexity of Homogeneous packing in the best case is: $O(\Delta \times m^2)$.

In the worst case, the tasks do not have affinities between them, so the total data weight remains the same and the number of packages only decreases by 1 at each iteration of the inner while loop. There are therefore m iterations before having only one package left. We note $a = \sum_i |\mathcal{D}(T_i)|$ the number of access to data on all the tasks, thus, whatever the number of fusion, the number of loads will be less than a. $\sum_i |\mathcal{D}(T_i)| \leq m \times \max_i |\mathcal{D}(T_i)|$, so $a \leq m \times \Delta$. We can rephrase "Find a package P_j such that $|\mathcal{D}(P_i) \cap \mathcal{D}(P_j)|$ is maximal" this way:

- For $P_i \in \mathbb{P}$ we read $\mathcal{D}(P_i)$
- For $P_j \in \mathbb{P}$ we read $\mathcal{D}(P_j)$
- For each of these two loops there are at most a iterations
- Combined this gives a complexity of $O(a^2)$

a is bounded by $m \times \Delta$. So we get a complexity of $O(m^2 \times \Delta^2)$ to find the package P_j. Knowing that there are m iterations until getting only one package we have the complexity in the worst case: $O(m^3 \times \Delta^2)$.

In linear algebra in particular, all tasks have a very similar data access pattern and the data shares are regular. Hence, in practice, the ceomplexity is often $O(\Delta \times m^2)$.

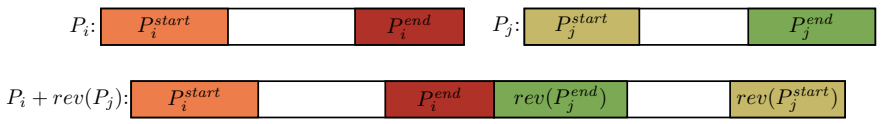


Figure 2: Flipping packages to improve HFP. Here we assume that the pair of sub-packages (P_i^{end}, P_j^{end}) is the one with the most shared input data, so that only P_j is reversed before merging packages.

Improving HFP with package flipping One problem may appear in the second phase of HFP, when we merge packages without taking care of the M bound: if P_i is merged with P_j, the merged package contains the tasks of P_i followed by the ones of P_j. However, the last tasks of P_i might have very little shared data with the first tasks of P_j, leading to poor data reuse when starting P_j. Hence, for each package P_i, we consider two sub-packages P_i^{start} and P_i^{end} containing the first and last tasks so that the weight of their input data is smaller than M but their cardinal is maximal, as illustrated on Figure 2. Then, we count the common input data of tasks P_i^{start} and P_i^{end} on one side, and P_j^{start} and P_j^{end} on the other side. We identify the pair with most common input data and selectively reverse the packages so that tasks in this pair of sub-packages are scheduled consecutively in the resulting package.

Package flipping requires to go through the set of tasks of two packages. In the worst case, the two packages together contain all of $\mathbb{T}$, so the complexity is $O(m)$. This complexity can be neglected compared to the original complexity of HFP.

Figure 3 shows an example of the tasks processing order of C on a 2D matrix multiplication with and without package flipping.

16	18	24	26	68	70	76	78	0	2
17	19	25	27	69	71	77	79	1	3
20	22	28	30	72	74	80	82	4	6
21	23	29	31	73	75	81	83	5	7
32	34	40	42	84	86	92	94	8	10
33	35	41	43	85	87	93	95	9	11
36	38	44	46	88	90	96	98	12	14
37	39	45	47	89	91	97	99	13	15
60	62	64	66	48	50	52	54	56	58
61	63	65	67	49	51	53	55	57	59

(a) Without package flipping

0	3	15	12	48	51	63	60	47	44
1	2	14	13	49	50	62	61	46	45
7	4	8	11	55	52	56	59	40	43
6	5	9	10	54	53	57	58	41	42
31	28	16	19	79	76	64	67	39	36
30	29	17	18	78	77	65	66	38	37
24	27	23	20	72	75	71	68	32	35
25	26	22	21	73	74	70	69	33	34
92	95	96	99	80	83	84	87	88	91
93	94	97	98	81	82	85	86	89	90

(b) With package flipping

Figure 3: Tasks computation's order with HFP with or without package flipping on a 10x10 matrix

4 Experimental evaluation

We present here the results of the experimental evaluation conducted to compare the strategy presented above.[1]

4.1 Settings

All above strategies have been implemented in the StarPU runtime system [3]. This allows to test them on a variety of applications described as sets of tasks. In order to test the performance of the strategies in different conditions (such as varying GPU memory size), we have used the ability to run StarPU code over the SimGrid simulator [6]. To make simulation times tractable, we have chosen characteristics of the simulated platform which are relatively small, to allow

[1] The code used to reproducibly obtain the results in this paper is available at `https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/europar2021`

observing locality effects even with a small input matrix: the PCI bandwidth is set to 350MB/s, and the GPU memory size evolves around 500MB. The scheduling algorithms receive the whole set of tasks of the application in a natural order (row by row for a matrix multiplication for instance), then outputs this same set of task in a new order, which is used in StarPU to process tasks on the GPU. We measure the obtained performance (in GFlop/s) as well as the total volume of data transferred between CPU and GPU.

We use four set of tasks for these experiments:

2D matrix multiplication To compute $C = A \times B$ in parallel, each task corresponds to the multiplication of one block-row of A per one block-column of B. Input data are thus the rows of A and columns of B.

3D Matrix multiplication All matrices (A, B, C) are tiled, and the computation of each tile of C is decomposed in multiple tasks, each of which requires one tile of A and one tile of B. Tiles of C are also used as input for all tasks on this tiles but the first one.

Task set from the Cholesky factorization We consider the tasks of the tiled Cholesky decomposition [1], but remove all dependencies, as we are interested only in independent tasks. This allows to have dependencies with some regularity, but more complex than the 2D or 3D matrix multiplication.

Randomized 2D matrix operation We consider the set of tasks and data from the 2D matrix multiplication, but with a random dependency pattern between tasks and data: each task requires one (random) block-row of A and one (random) block-column of B. This allows to test our algorithms on an unstructured dependency graph.

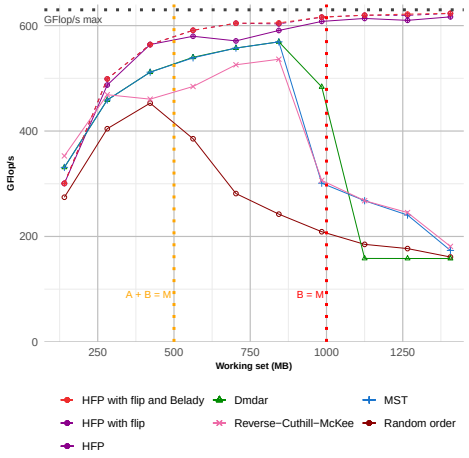
We use the four scheduling heuristics presented above, together with a random task ordering as a baseline. We use the default LRU eviction policy, but also consider the optimal eviction policy, denoted by Belady, presented above. This policy is available only for static heuristics (as it requires complete task order from the beginning), such as RCM, MST and the proposed HFP. For the sake of clarity, we present only the result of the Belady eviction policy on HFP.

4.2 Results on the 2D matrix multiplication

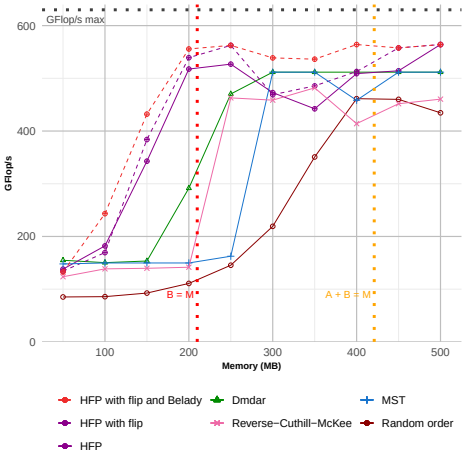
On Figure 4, we plot the performance of each scheduling heuristic when varying either the size of the problem, or the size of the available memory. We also show the quantity of data transfers and the execution time. On these graphs, the dotted horizontal black line represents the maximum GFlop/s (630) that the GPU can achieve when processing elementary matrix product (without I/Os) and is our asymptotic goal. The red dotted vertical line denotes the situation when the GPU memory can fit exactly only one of the two input matrices, and the orange line denotes the situation when it can accommodate both input matrices.

A pathological matrix size for non-data aware algorithms When using a random task order, performance drops right after the orange line. Indeed, when the two input matrices do not fit in memory, the random order keeps triggering data reloads. The DMDAR, MST and RCM heuristics also get pathological behavior after the red line. Indeed, they tend 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, which is a well-known pathological case for the LRU eviction policy.

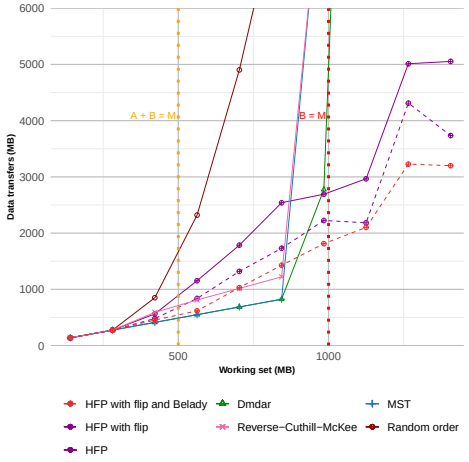
What happens? First, we have to understand how the LRU works when multiplying matrix. We multiply A by B to get C. Consider that we use the rows of A and the columns of B. So



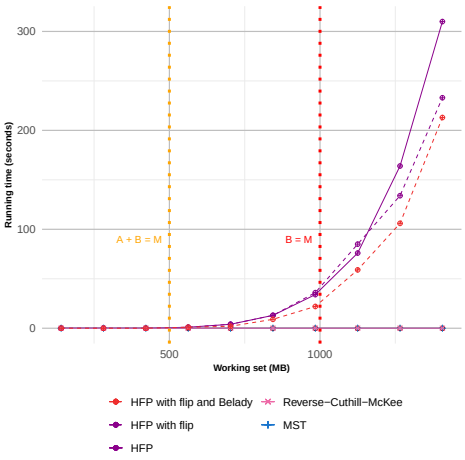
(a) GPU memory size fixed to 500MB, varying working set size.



(b) Working set size fixed to 422MB, varying GPU memory size.



(c) Data transfers.



(d) Running time.

Figure 4: 2D matrix multiplication.

for small matrices we can for example load all of B, a row of A and a piece of C to write the result in it. We compute the first row of C, then we will want to load the second row of A to be able to compute the second row of C. So the LRU will evict the last data used, the first row of A. It goes without problems. On Figure 4a, after 1000MB, neither A nor B fits in memory. The scheduler is therefore forced to load a few columns of B, a row of A and a block of C. It compute the first row of C. Unfortunately it could not load all the columns from B, so when we want to compute a block where all the data are not on memory, it has to evict the first column from B (the oldest therefore) in order to load the column of B it needs. But when we go to the computation of the second row of C, we need the first columns of B that we just evicted. It must therefore again evict the last columns of B. This generates many additional data transfers. DMDAR also treat task row by row but when going onto the next row, it choose to compute the tasks that already have data loaded, so using the last columns of B. However once these tasks are done it will treat the first tasks of the row, thus generating a large number of transfers. So all the algorithms treating tasks row by row or column by column will suffer from this is well-known pathological case of LRU. HFP aim to avoid that.

Performances of HFP The HFP heuristic gets performance very close to ideal. Indeed, it tends to gather tasks that compute a square part of C that require parts of A and B that can fit in memory size M. This allows to execute a lot of tasks with very few data to load. The package flipping and Belady variants further improve the results.

Here are the percentage of improvement of HFP with package flipping and Belady over the other heuristics, averaged on the ten points:

Reference algorithm	Random	MST	RCM	DMDAR	HFP	HFP with flip
Improvement	103.6%	42.9%	47.4%	43.8%	1.9%	0.1%

Varying memory size Figure 4b shows the dual view of Figure 4a: the working set is now set to 422MB and we simulate varying amounts of available GPU memory. The measurements at 500MB on Figure 4b are the same as the measurements at 422MB on Figure 4a. We can observe the same results as on 4a but reversed: when the available memory is smaller than the working set, heuristics get pathological behavior. Since we strongly reduce the amount of available memory, we get a more restrictive situation, and package flipping provides a large improvement here (4% versus 1.8% on Figure 4a). Indeed, when memory is more restricted, the second phase of HFP (without bound on package size) is more important, and it is essential to optimize the transition between packages. Moreover HFP with Belady improve GFlop/s by 7% compared to its variant without it. It is all the more important to evict data correctly when the number of data available in memory is very low.

Belady improvement We can see on Figure 4a that the differences between HFP and HFP with Belady are very slim, HFP with Belady is only 1.5% better. However on Figure 4b, it is 7.7% better. These are situations where the memory is very limited, we cannot fit a whole matrix on the GPU memory. Let's say that we want to process a set of tasks that uses the first 8 rows of A and the first 8 columns of B. We first compute the block of C using the first row and the first column. Let's say that we can compute the first 4 rows of tasks before the memory is full. Then LRU will evict starting from data from the first row of A and the first column of B. But when we need to compute the next tasks that are tasks on the same columns but on the last 4 rows of A, we need data that we just evicted. On the other hand, if we use Belady, in the same situation it would evict data from the first few rows of A, not used again before the last tasks. Thus in this situation Belady saves us a lot of data transfers. For example for the fourth point

of Figure 4b, the number of data transfers without Belady is 84, with belady it is 46. We can conclude that if a scheduler knows all the tasks before the start of the execution, using Belady is undoubtedly beneficial.

Data transfers and execution time Figure 4c shows the quantity of data transfers done in MB. It uses the same parameters as the Figure 4a. By looking at these two figures we can see that data transfers increase drastically when GFlop/s decrease.

On Figure 4d we can see that only HFP has a lot of execution time. This happen when there are a lot of tasks. The time it takes to compare each data of each package is exponential with the number of tasks.

4.3 Results on the 3D matrix multiplication

On Figure 5, we plot the performance and amount of data transfers for all heuristics on the 3D matrix multiplication. On this set of tasks, matrix C now plays a role in affinities, which is why we added a vertical green dotted line to denote the situation when all A, B, and C matrices fit in memory.

Performances of MST MST keeps ordering tasks along the rows of C, and thus still gets pathological performance when memory can not fit matrix B. This is confirmed on Figure 5c: the number of loads gets dramatically high.

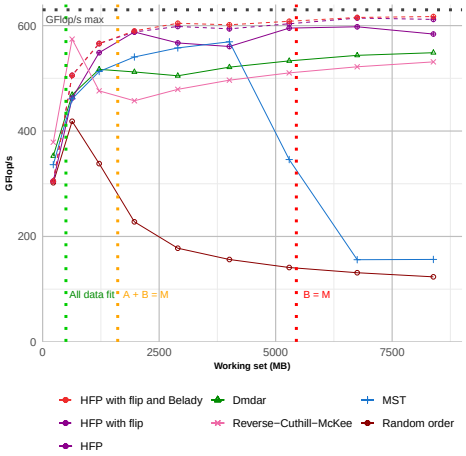
DMDAR and RCM RCM and DMDAR, however, do not have the same problem. RCM (resp. DMDAR) computes tasks along columns (resp. rows) of C but alternates between tasks of a few consecutive columns (resp. rows). This allows them to improve data reuse: Figure 5c shows that they exhibit a limited number of transfers, even with a large working set.

Performances of HFP HFP keeps gathering tasks forming a square part of C, which provides better locality. Here are the percentages of average improvement of HFP with package flipping and Belady over the other heuristics:

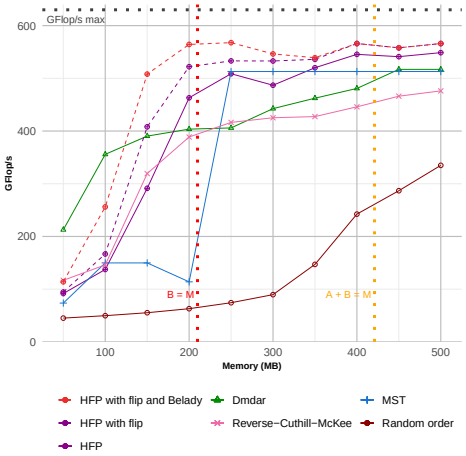
Reference algorithm	Random	MST	RCM	DMDAR	HFP	HFP with flip
Improvement	148.7%	37.9%	13.3%	11.4%	4.3%	0.5%

As the 3D matrix multiplication already exhibits a better data locality than the 2D multiplication, the differences in performance between heuristics is less pronounced than on Figure 4a, but HFP is still better on average. It is worth noticing that HFP without Belady gets higher performance than MST, RCM and DMDAR, even if it exhibits a similar number of transfers. The latter heuristics indeed tend to periodically require a sudden burst of data loads, while HFP tends to require loads that are nicely distributed over time, and thus well overlapped with computation.

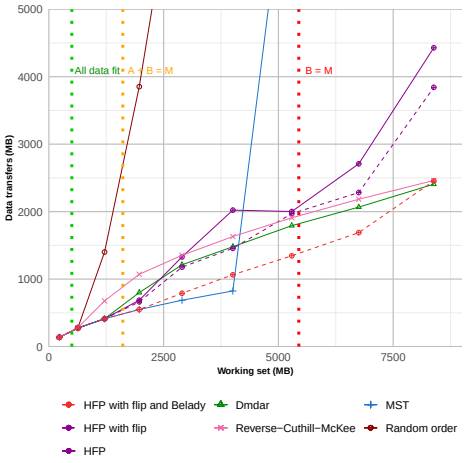
Number of loads We can see on Figure 5c that the number of loads of DMDAR and RCM are consistent with what observe on Figure 5a. However we can see that HFP got better performance than DMDAR but also more data transfers. DMDAR has no long-term visibility. It will compute task that have data already loaded, so more often than not it will compute tasks row by row and insert tasks from other rows in between. So when the working set is too large for the memory size, it can have a lot of loads at once. Hence the distribution of the loads in time is reducing the GFlop/s but does not increase the number of data transfers. On the other hand, HFP groups the tasks in packages of size M which allows to make homogeneous packages that fit on memory.



(a) GPU memory size fixed to 500MB, varying working set size.



(b) Working set size fixed to 1250MB, varying GPU memory size.



(c) Data transfers.

Figure 5: 3D matrix multiplication.

So we have to load data each time we go onto an other packages. Which distribute transfers a lot. So even if there are more data transfers, the GFlop/s are not affected because the data transfers are spread out and we can therefore compute tasks during the transfers. We can also observe that Belady reduce the number of data transfers by 32.5% from HFP without Belady.

Varying the size of the memory On Figure 5b, we can see that DMDAR is better than HFP for the two smallest memory sizes. For very small memory, HFP has a lot of packages, so it cannot build packages sharing a lot of data and can only order packages between them which result in putting one after another tasks that won't share a lot of data.

4.4 Results on the task set of the Cholesky factorization

Random order, RCM and MST Figure 6a present the performance obtained by the heuristics on the set of tasks of the Cholesky decomposition. We notice that these three algorithms get pathological performance once the whole matrix cannot fit the memory. They indeed do not manage to reuse more than one tile between consecutive tasks, thus entailing a lot of tile reloads.

Favorable natural order for DMDAR DMDAR has better results than HFP (with or without Belady) for a working set inferior to 1.5 times the memory. DMDAR indeed takes advantage of the actual task submission order of the Cholesky algorithm, which starts with tasks which require few input data (POTRF and TRSM kernels). Meanwhile, it can load data for the subsequent tasks with more input dependencies (GEMM kernel). HFP, on the contrary, does not pay attention to the task submission order, and aims for data sharing as much as possible. It will thus introduce a lot of GEMM tasks at the beginning of the execution, and is thus impacted by the loading time. As the working set increases, however, HFP with the Belady eviction achieves good performance.

Performances of HFP The average improvement of HFP with flip and Belady over the other heuristics is as follows:

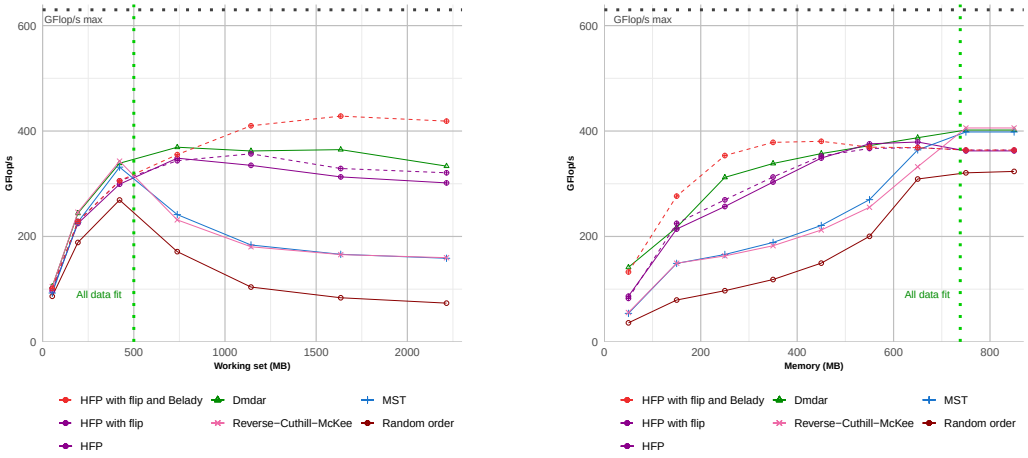
Reference algorithm	Random	MST	RCM	DMDAR	HFP	HFP with flip
Improvement	127.1%	60.4%	57.1%	6.2%	17.2%	13.2%

Varying the size of the memory As we saw on Figure 6a, we can see on Figure 6b that once all the data fit on memory, DMDAR, MST and RCM are the best algorithms. This is explained by the fact that HFP does not recognize that all the data fits in memory and therefore seeks to make packages which force the loading of certain data several times.

Data transfers Again we can see on Figure 6c that the number of data transfers are equivalent for HFP Belady and DMDAR whereas the number of GFlop/s for HFP with Belady are greater. Again the distribution of loads over time is better for HFP.

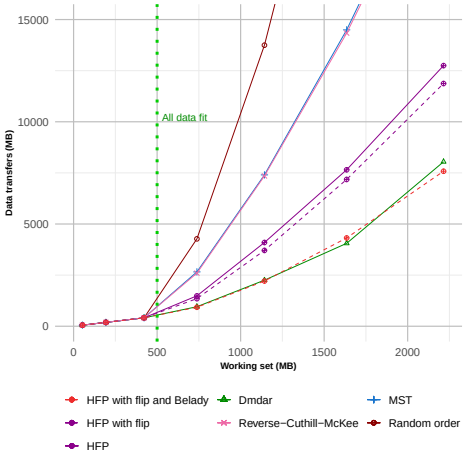
4.5 Results on the randomized 2D matrix operation

The randomization of data dependencies of the 2D matrix multiplication, shown on Figure 7a, has an interesting impact on the results previously discussed from Figure 4a. In this randomized setting, MST does not manage to find task affinities any more: it is only able to ensure that a common data is shared between two consecutive tasks, but since it does not benefit from the natural order of task arrival anymore, its performance drops as soon as the two matrices



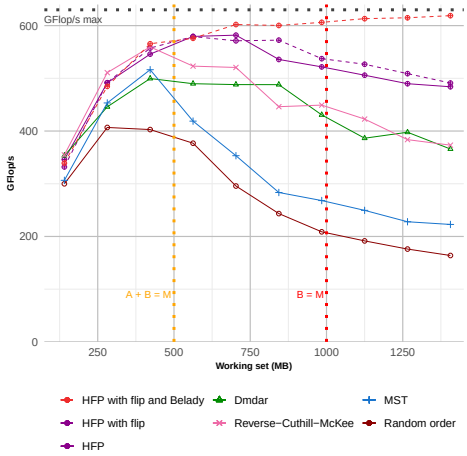
(a) GPU memory size fixed to 500MB, varying working set size.

(b) Working set size fixed to 740MB, varying GPU memory size.

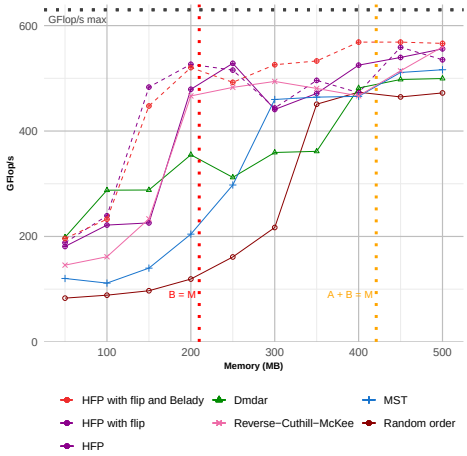


(c) Data transfers.

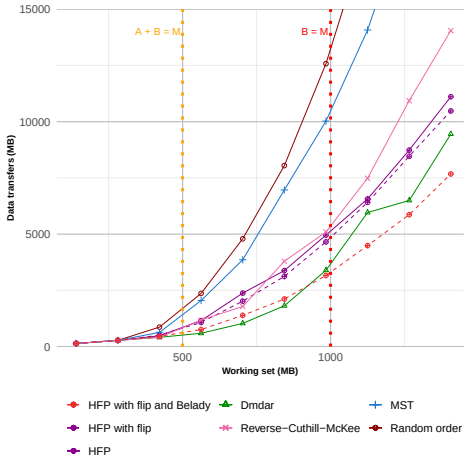
Figure 6: Task set of the Cholesky factorization.



(a) GPU memory size fixed to 500MB, varying working set size.

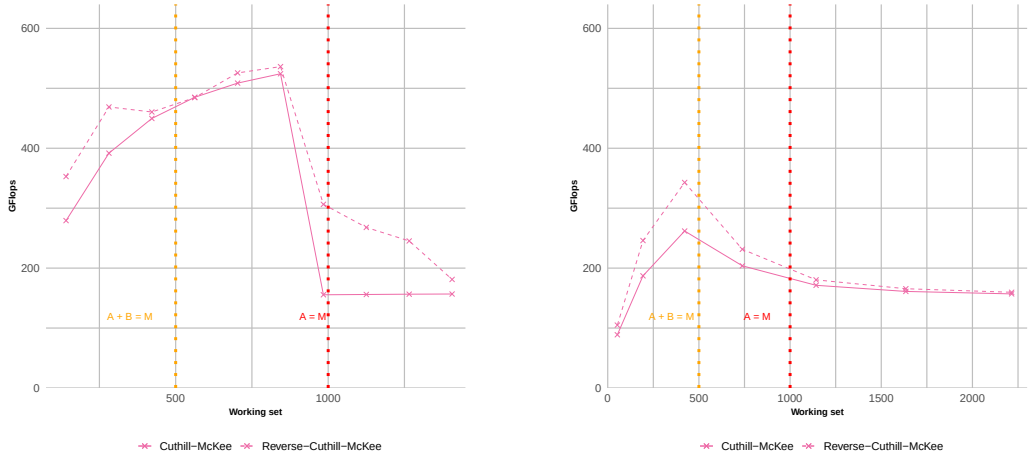


(b) Working set size fixed to 422MB, varying GPU memory size.



(c) Data transfers.

Figure 7: Randomized 2D matrix operation.



(a) On the 2D matrix multiplication.

(b) On the task set of the Cholesky factorization.

Figure 8: Comparison of Cuthill-McKee and Reverse-Cuthill-McKee.

no longer fit into memory. RCM and DMDAR, however, get more sustained performance: the randomization of dependencies actually decreases the effect of the classical LRU pathological case, since they do not tend to execute tasks rows by rows any more. HFP, however, can more nicely find task affinities thanks to its global overview. The addition of the Belady eviction allows to keep excellent performance, thanks to its insight into the ordering of the randomized tasks.

Performances of HFP Its average improvement over the other heuristics is as follows:

Reference algorithm	Random	MST	RCM	DMDAR	HFP	HFP with flip
Improvement	103.2%	70.3%	23.7%	29.4%	10.6%	8.8%

We can also note that when the working set grow, HFP without Belady loses performance. On the other hand, with Belady it stay close to GFlop/s max. Indeed, if the data are random, HFP with LRU can no longer rely on the favorable natural order of tasks in 2D. Belady therefore allows improvements in more general cases as well.

In summary, the simple greedy DMDAR heuristic exhibits robust performance, but HFP, and particularly with the package flipping and Belady improvements, exhibit better performance, including in the case of a tight memory.

5 Conclusion and Future Work

To take the best performance out of GPUs, it is crucial to avoid moving data as much as possible. We provided in this paper a formalization of the problem of ordering independent tasks sharing input data in order to minimize the amount of data transfers, and show that this problem is NP-complete. We also exhibit an optimal eviction scheme, based on Belady's rule. We adapted three heuristics for the ordering problem, based on the state of the art, and compared them with a new algorithm gathering tasks with similar input data into packages of increasing size, called

HFP. We also present an improvement of HFP based on package flipping. All four ordering strategies have been implemented in the StarPU runtime and extensively tested on various set of tasks. In all cases, the proposed HFP heuristic provides significant speedups. For instance, it allows on average a 43% (resp. 11%) improvement for 2D (resp. 3D) matrix multiplication. HFP is very relevant and obtains important speedups particularly in the case when the memory is very constrained compared to the size of the total working set. We also notice that in some situations, HFP ends up with as many data transfers than another heuristic, but with better performance, which shows that HFP is also good in distributing data transfer over time to increase transfer/computation overlap. Studying this final problem (minimizing computation time with overlap) is one for our future directions. We also plan to focus on the very beginning of the execution, where it is crucial to first schedule tasks with few input data. On a longer term, we want to tackle the general case with tasks sharing not only input data, but with inter-task dependencies, as well as targeting multi-GPU platforms, for which our approach with packages seems particularly well suited.

Acknowledgments

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

References

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

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

[3] 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, Special Issue: Euro-Par 2009 23 (2011)

[4] Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5(2) (1966)

[5] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.: PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. Computing in Science and Engineering 15(6), 36–45 (Nov 2013)

[6] Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: Versatile, scalable, and accurate simulation of distributed applications and platforms. Journal of Parallel and Distributed Computing 74(10) (Jun 2014)

[7] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009)

[8] Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th National Conference. ACM '69 (1969)

[9] Denning, P.J.: The working set model for program behavior. Communications of the ACM 11(5), 323–333 (1968)

[10] Feder, T., Motwani, R., Panigrahy, R., Seiden, S., van Stee, R., Zhu, A.: Combining request scheduling with web caching. Theoretical Computer Science 324(2) (2004)

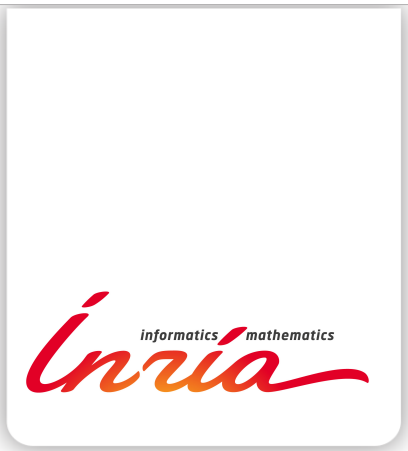
[11] Garey, M.R., Johnson, D.S.: Computers and Intractability, A Guide to the Theory of NP-Completeness. W.H. Freeman and Co, London (UK) (1979)

[12] Gavril: Some np-complete problems on graphs. In: Proceedings of the 11th conference on Information Sciences and Systems. pp. 91–95 (1977)

[13] Kaya, K., Uçar, B., Aykanat, C.: Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories. J. Parallel Distributed Comput. 67(3) (2007)

[14] Michaud, P.: (yet another) proof of optimality for min replacement (Oct 2007)

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



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399