Locality-Aware Scheduling of Independant Tasks for Runtime Systems

Maxime Gonthier, Loris Marchal, Samuel Thibault

Locality-Aware Scheduling of Independant Tasks for Runtime Systems

Maxime Gonthier[1], Loris Marchal[1], and Samuel Thibault[2]

[1] LIP, CNRS, ENS de Lyon, Inria & Université Claude-Bernard Lyon 1,
firstname.lastname@ens-lyon.fr
[2] LaBRI, University of Bordeaux, CNRS, Inria Bordeaux – Sud-Ouest
samuel.thibault@u-bordeaux.fr

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 independant) 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 tasks 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.

Keywords: Memory-aware scheduling, Eviction policy, Tasks sharing data, Runtime systems.

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 [4] or StarPU [2]. 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 a 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 his problem, and propose a new one based on gathering tasks with similar data patterns into packages (Section 3).
- We perform extensive experimental evaluations to study both the amount of data transfers of these algorithms on sets of tasks coming from actual applications and the performance obtained when plugging these algorithms into the StarPU runtime (Section 4). Overall, our evaluation shows that

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.

Related work.

- general stuff on data movement and memory limitation?
- paper with MST [12]: the closest, same pb of ordering independent tasks sharing data, but for computation on multicore CPU. Multiple PUs (parallel processing) contrarily as our study, so they have two problems: grouping and ordering. The proposed heuristic MST, is described below and compare to others in our experiments.
- task sharing files: already studied in a distributed context (grid): cf paper from Bora et al.[10]
- By some aspects, our work is close to some studies in cache optimization, which consists in reordering requests to optimize cache reuse [8]. However, contrarily to these studies, we know the whole set of tasks (or requests) and not only a subset, and we are allow to totally re-order it.

2 Problem modeling and complexity

We consider the problem of scheduling independent tasks on a GPU of memory size M. As proposed in previous work [10], 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.

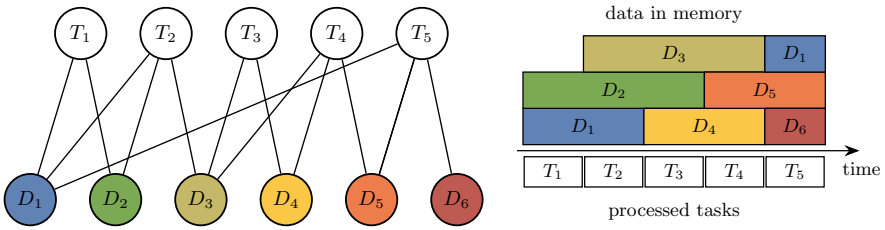


Fig. 1: Example with 5 tasks and 6 data, with a memory holding at most $M = 3$ data. The graph of input data dependencies is shown on the left. The schedule on the right corresponds to processing the tasks in the natural order with the following eviction policy: $\mathcal{V}(1) = \mathcal{V}(2) = \emptyset$, $\mathcal{V}(3) = \{1\}$, $\mathcal{V}(4) = \{2\}$, $\mathcal{V}(5) = \{3, 4\}$. 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 n steps, each steps 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:

$$L(t) = \begin{cases} \mathcal{D}(T_{\sigma(0)}) & \text{if } t = 0 \\ L(t) = (L(t-1) \backslash \mathcal{V}(t)) \cup \mathcal{D}(T_{\sigma(t)}) & \text{otherwise} \end{cases}$$

Our memory limitation can then be expressed as $|L(t)| \leq M$ for each step $t = 1, \ldots, n$. 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 occur 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 [11]: 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 [3]. 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 the problem* MinLoadsForTasksSharingData. *We denote by MIN the thrifty eviction policy that always evicts a data whose next use in σ is the lastest (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 tast 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 $T_{\sigma(t)}^{(1)}, T_{\sigma(t)}^{(2k)}$ (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 $T_i^{(1)}, T_i^{(2k)}$ 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 $T_i^{(j)}$. 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 $T_i^{(j)}$ belongs to V: during the first k tasks, their next occurence 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 in the furthest in the futre.

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) 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 [9]. 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 (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 motivate the use of the Reverse-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 [7]. 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 "neighbors" 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. The adapation 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

Maximum Spanning Tree (MST) As outlined in the related work, another heuristic has been propose to order tasks sharing data to improve data locality [12]. The authors of this study propose to build a Maximum Spanning Tree in the graph G^T using Prim's algorithm [6] 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.

Deque Model Data Aware Ready (DMDAR) DMDA or "Deque Model Data Aware" is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing elements in the StarPU runtime. It takes data transfer time into account and schedules tasks where their completion times is expected to be minimal [1] (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 element, 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 2). 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 2 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 build packages of tasks, denoted by $P_k \in \mathbb{P}$, which are stored as lists of tasks. 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 anymore data movement. We have proven that building the minimum number of packages is NP-complete, hence we rely on a greedy heuristic to build them, described in Algorithm 3. We initially consider 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 input size bound. The intuition is to create meta-packages that express the data affinity between packages. Note that we do not modify the order of tasks within packages when merging them, hence keeping the good data locality inside packages. Finally, the last remaining package is the list of tasks in the schedule.

Algorithm 3 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
```

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

$(\frac{m}{2^i})^2 * \Delta * 2^i = \frac{m^2 * \Delta}{2^i}$. If we sum the i iterations we get:

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

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

In the worst case, the tasks do not have affinities between them, so the total data weight remains the same and the number of packets only decreases by 1 at each iteration. There is 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. $\sum_i |\mathcal{D}(T_i)| \leq m * \max_i |\mathcal{D}(T_i)|$, so $a \leq m * \Delta$. The computation line 11 can be approximated by a reading of all the data of the two packages that we compare: $\mathcal{D}(P_i)$ and $\mathcal{D}(P_j)$. We can rephrase lines 9, 10 and 11 with this knowledge. Indeed we can say that we do:

- 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 $m * \Delta$. So we get $O(m^2 * \Delta^2)$ for the lines 9, 10 and 11. So knowing that there are m iterations of the while loop line 5 we get: $O(m^3 * \Delta^2)$.

In practice, we often find ourselves in the perfect case. In linear algebra in particular, the matrix are often square and the blocks of the same size, with very similar data. In the same way generally the data shares are regular in linear algebra. So except for a constant, the behavior of the algorithm will be like in the best case.

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 follows by the ones of P_j. However, the last tasks of P_i might have little shared data with the first tasks of P_j, leading to poor data reuse. 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 package so that tasks in this pair of sub-packages are scheduled consecutively in the resulting package.

More precisely, we perform as follows:

1. We want to merge P_i and P_j
2. We create 2 sub-packages of tasks for P_i and P_j:
   (a) If $W(P) > M$, we create two sub-packages containing the first and last k tasks of P such as $weight(\sum^k T) \leq M$ and k is maximal.

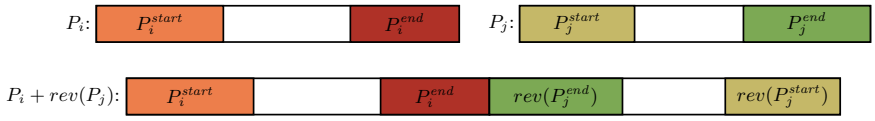


Fig. 2: Flipping packages to improve HFP. Here we assume that the pair of sub-packages (P_i^{end}, P_j^{end}) is the one with the most shared input data.

(b) Else if only one package exceed M we consider the small package as a whole and create two sub-packages for the big one as in the previous step.
(c) Else if $W(P) \leq M$ the sub-packages correspond to the 2 packages used in the previous iteration to build P_i and P_j.

3. We compare the weight of the common data of these sub-packages
4. We reverse the order of tasks in P_i and or P_j so that the last tasks of (possibly reversed) P_i share the most data with the first tasks of (possibly reversed) P_j.

This algorithm requires to go through the set of tasks of two packages. In the worst case, the two packages together contain the set of tasks, so the complexity is $O(m)$. This complexity is added to the complexity of HFP, it therefore does not add complexity, it can therefore be neglected.

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 U order

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 U order

Fig. 3: Tasks computation's order with HFP with or without U order on a 10x10 matrix

HFP with Belady Lastly an other variant of HFP used in this article is to use an adaptation of Belady's eviction strategy. Thanks to StarPU we can prefetch data, so we cannot define in advance a data eviction order like Belady does. In addition, several tasks may be processed at the same time, which makes making an eviction order in advance obsolete. What we can do is, when a data needs to be evicted, choose the one that will be used in the longest time.

All non-dynamic algorithms (including MST and CM) can take advantage of Belady. We did not add it to them on the experiments because there were no sufficient improvement in our applications.

4 Experimental evaluation

4.1 Settings

The implementation used for all the results discussed in this paper is available: `https://gitlab.inria.fr/mgonthie/locality-aware-scheduling.git`.
The implementation is done in StarPU using simulations thanks to Simgrid 3.26 [5]. To make simulation times tractable, we have chosen characteristics of the simulated platform which are relatively small, to allow 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 algorithm receives 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. StarPU then uses Simgrid to simulate the execution of tasks on a platform composed of a single GPU. We then obtain the achieved performance in terms of GFlop/s, and the number of data transfers required during execution. The evaluated application algorithms are:

Simplified (2D) matrix multiplication To compute $C = A \times B$ in parallel, each task is responsible for computing one tile of C, thus requiring the load of one slice of A and one slice of B. Data affinities are thus only on the slides of A and B.

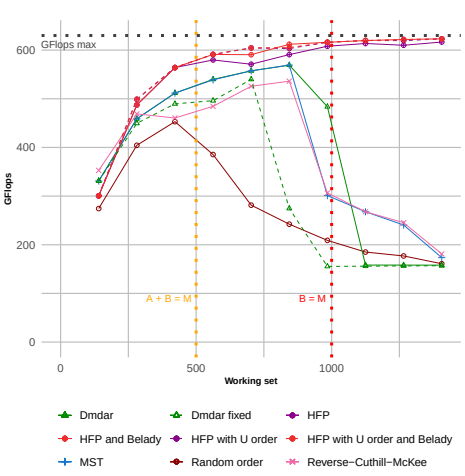
Classical (3D) Matrix multiplication The computation of each tile of C is decomposed in multiple tasks, each of which requires the load of one tile of A, B, and C. Data affinities are thus on all tiles of A, B, and C.

Cholesky factorization Since the algorithms presented here (except DMDAR) do not manage dependencies, this is the set of tasks of the Cholesky factorization (POTRF, TRSM, SYRK and GEMM) without dependencies.

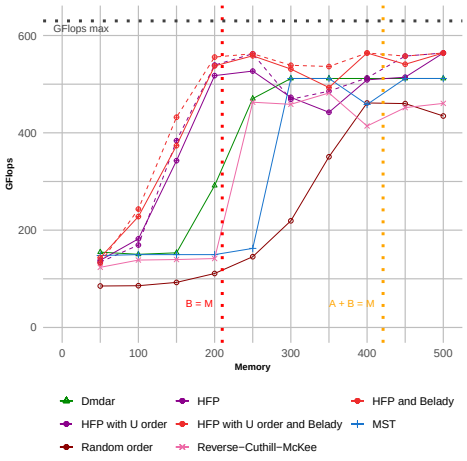
Randomized 2D matrix multiplication The sets of task and data are the same as the matrix multiplication in 2D but the data used by a task are chosen randomly.

4.2 Results on the 2D matrix multiplication

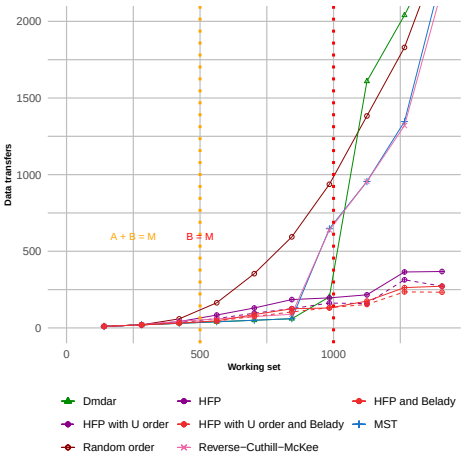
On the resulting graphs in Figure 4a, the dotted horizontal black line represents the maximum GFlops (630) that the GPU can achieve, thus the asymptotical goal. The red dotted vertical line at 1000MB denotes the situation when the GPU memory can fit exactly only one of the two matrices, and the orange line denotes the situation when it can fit the two matrices.



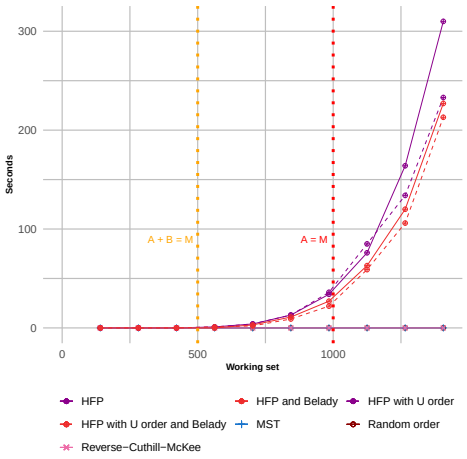
(a) Varying the size of the working set



(b) Varying the size of the GPU's memory with N=15 (422MB)



(c) Varying the size of the working set

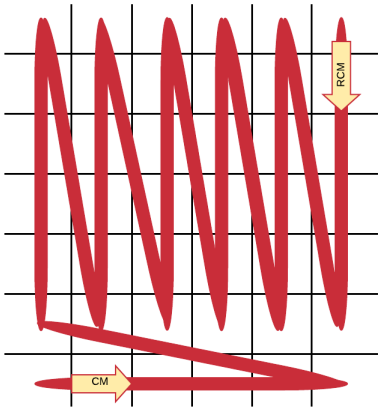


(d) Looking at time execution

Fig. 4: Evolution of GFlops on a 2D matrix multiplication

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 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. For CM the processing order is the first row of C, then column by column as you can see here: This allows to reuse the same slice of matrix A for the tasks that compute



tiles of the same row of C, but requires loading the whole matrix B. When the GPU cannot fit the whole matrix B, when processing the end of a row of C the left part of B is getting evicted, and when proceeding with the beginning of the next row of C, the left part of matrix B has to be reloaded, and the next part be evicted, only to be reloaded immediately after this, and so on with perpetual reloading. This is the well-known pathological case of LRU.

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 columns of A and the rows of B. So for small matrices we can for example load all of A, a row of B and a piece of C to write the result in it. We calculate the first row of C, then we will want to load the second line of B to be able to calculate the second line of C. So the LRU will evict the last data used, the first line of B. So it goes without problems. We can see that performances falls right before the line and not after the line because a block of C need to be loaded in order to do the computation too. After 1000MB, neither A nor B fits in memory. The scheduler is therefore forced to load a few columns of A, a row of B and a block of C. It compute the first line of C. Unfortunately it could not load all the columns from A, so when we want to compute a block where all the data are not on memory, it has to evict the first column from A (the oldest therefore) in order to load the column of A it needs. But when we go to the computation of the second line of C, we need the first columns of A that we

just evicted. It must therefore again evict the last columns of A. This generates many additional data transfers. DMDAR also treat task line by line but when going onto the next line, it choose to compute the tasks that already have data loaded, so using the last columns of A. However once these tasks are done it will treat the first tasks of the line, thus generating a large number of transfers. So all the algorithms treating tasks line by line 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 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, and avoids the pathological case mentioned before. The U order and Belady variants further improve the results.

Here are the percentage of improvement of HFP with U order and Belady over the other heuristics, on average on the ten points:

Algorithms	Random	MST	RCM	DMDAR	HFP	HFPU	HFP + Belady
Improvement	103.6%	42.9%	47.4%	43.8%	1.9%	0.1%	0.3%

Varying memory size Figure 4b shows the dual view: the working set is now set to 211MB and we simulate varying amounts of available GPU memory. The mesurements at 500MB on Figure 4b are the same as the measurements at 420MB 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 Order U provides a large improvement here (4% versus 1.8% on figure 4a). Indeed, when memory is more restricted, HFP builds more packages and thus it becomes more important to optimize the linking between two packages. The Belady eviction also provides a slight improvement, since it avoids the LRU trap.

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 the figure 4b, it is 7.7% better. These are situations where the memory is very limited, we can't fit a whole matrix on the GPU memory. The first tasks processed with HFP's order are on the first 8 line and on the columns 1 to 4 and 11 to 15 of C. It is a package done by HFP before the removal of the package size limit. What happen for HFP with LRU is that for the first set of task mentionned we can process a total of 64 task with only 8 lines and 8 columns loaded. That corresponds to 240 data, in other word 217.9 MB. Let's say we have a memory of 200MB. Thus LRU will evict starting from data from the first line 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 8 lines, 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 lines of A, not used again before the last tasks. Thus in this situation Belady saves us a lot of data transfers.

0	3	15	12	185	188	200	197	181	184	63	60	48	51	201
1	2	14	13	186	187	199	198	182	183	62	61	49	50	202
7	4	8	11	192	189	193	196	209	212	56	59	55	52	203
6	5	9	10	191	190	194	195	210	211	57	58	54	53	204
31	28	16	19	161	164	176	173	216	213	32	35	47	44	205
30	29	17	18	162	163	175	174	215	214	33	34	46	45	206
24	27	23	20	168	165	169	172	217	220	39	36	40	43	207
25	26	22	21	167	166	170	171	218	219	38	37	41	42	208
85	88	121	124	136	133	120	117	224	221	105	108	104	101	96
86	87	122	123	135	134	119	118	223	222	106	107	103	102	95
92	89	128	125	129	132	113	116	137	140	112	109	97	100	94
91	90	127	126	130	131	114	115	138	139	111	110	98	99	93
78	81	160	157	152	149	148	145	144	141	64	67	75	72	76
79	80	159	158	151	150	147	146	143	142	65	66	74	73	77
83	82	156	155	154	153	177	178	179	180	68	69	70	71	84

Fig. 5: Task's processing order

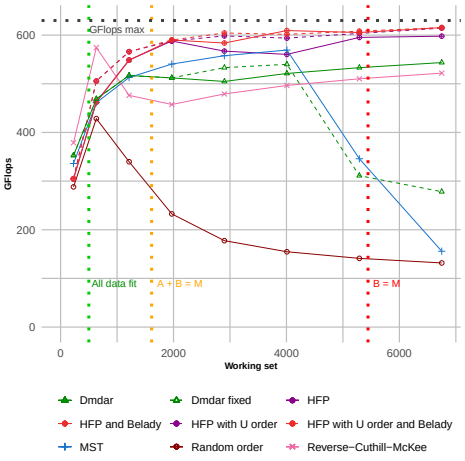
Indeed the number of data transfers for this case without Belady is 84, with belady it is 46. We can conclude that if a scheduler knows all the tasks before the start of the execution, using belady is undoubtedly beneficial.

4.3 Results on the 3D matrix multiplication

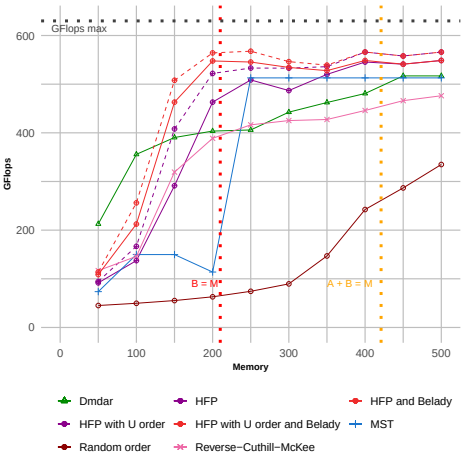
Differences 2D-3D On a 3D matrix multiplication, each task only need three data, one from the row of A, on from the column of B and the block of C. This is why we add a green dotted line here representing a working set size for which all the data needed for the computation fit on memory, so matrix A, B and C.

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. Inside a line of 10 task for example it first compute tasks alternating from blocks 1 and 5, then 2 and 6, 3 and 7 and so on. So once we can't fit a full matrix on the seventh point, the same problem as mentionned in section 4.2 happen and performances drop. This is confirmed on Figure **??**: the number of loads gets dramatic.

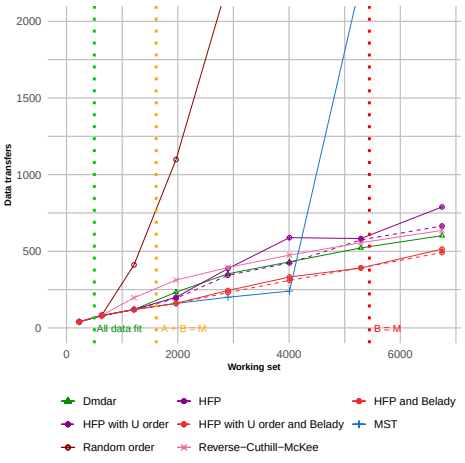
DMDAR and RCM RCM and DMDAR, however, do not fall into this trap any more. RCM (resp. DMDAR) computes tasks along columns (resp. rows) of C but alternates between tasks of a few consecutive columns (resp. rows). This allows them to improve data reuse, Figure **??** shows that they exhibit a limited number of transfers, even with a large working set. i can vary in function of the working set size. Once some data need to be evicted, it will be a column of B. So in the worst case we need to evict a column of B and re-load it for each new column of C we want to compute. DMDAR compute task row by row but also put in between tasks from a row above or under the current row in order to re-use data. We can see on the fiure **??** that it number of data transfers are very



(a) Varying the size of the working set



(b) Varying the size of the GPU's memory with a 210,9375 MB working set



(c) Varying the size of the working set

Fig. 6: Evolution of GFlops on a 3D matrix multiplication

close from those of RCM. These orders allows RCM and DMDAR to minimize their data transfers once the working set is too large and thus keeping constant GFlops after the red line.

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 U order and Belady over the other heuristics:

Algorithms	Random	MST	RCM	DMDAR	HFP	HFPU	HFP + belady
Improvement	132.1%	26.3%	12.9%	11.2%	4.1%	0.5%	1.8%

The differences are less important than on figure 4a, but HFP is still better on average. It is worth noticing on Figures **??** and **??** 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 required loads that are nicely distributed over time.

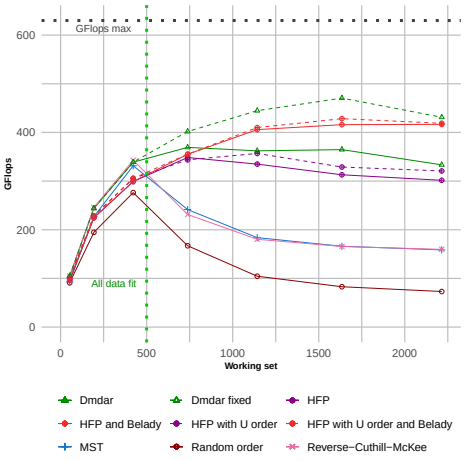
Number of loads We can see on the figure **??** that the number of loads of DMDAR and RCM are consistent with what observe on figure **??**. However we can see that HFP got better performances than DMDAR but also more data transfers. DMDAR has no long-term visibility. It will compute task that have data already loaded, so more often than not it will compute tasks line by line and insert tasks from other lines 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 GFlops 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. 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 GFlops 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 We can see that Dmdar is the two smallest memory size. For very small memory, HFP has a lot of packages, so it can't 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 any data.

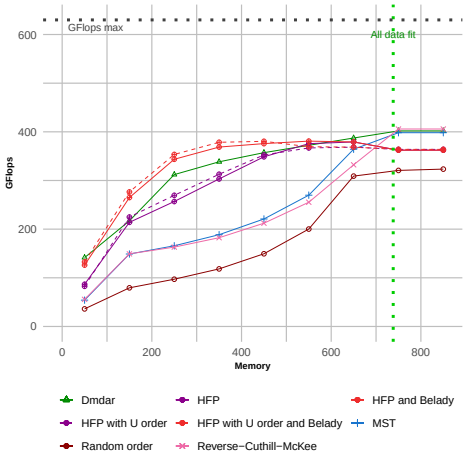
4.4 Results on the Cholesky factorization

The Cholesky factorization operates on a triangular matrix and exhibits a pyramidal set of task affinities with varying sets of dependencies.

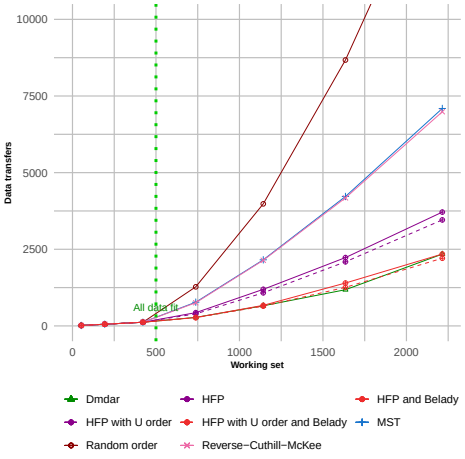
Random order, RCM and MST As can be seen on Figure 7a, these three algorithms get pathological performance once the whole matrix cannot fit the memory. They indeed do not manage to reuse more than one tile between consecutive tasks, thus entailing a lot of tile reloads.



(a) Varying the size of the working set



(b) Varying the size of the GPU's memory with a 738,28125 MB working set



(c) Varying the size of the working set

Fig. 7: Evolution of GFlops on a cholesky factorization

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 gets advantage of the actual task submission order of the Cholesky algorithm, which starts with POTRF and TRSM tasks which do not require many data. In the meanwhile, it can load data for the subsequent GEMM tasks. 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 as soon as the beginning of the execution, and thus get hit by the load delay. As the working set increases, however, HFP with the Belady eviction substains good performance. In these situations all the data or a big percentage of the data fit in memory. DMDAR start with POTRF tasks, which therefore produces TRSM data. Then continues with TRSM tasks which then only have one data to load. Also once the TRSM tasks are done, it produced data for the GEMM tasks which further reduce the number of loads needed. HFP wants to prioritize data sharing as much as possible. It will therefore group together the GEMM tasks because they use a greater number of data than TRSM and POTRF and therefore share more data. But if TRSM shares a data, it represents half the number of data of this task. This is not recognized by HFP if more data are shared elsewhere, even if the share percentage is lower. Thus HFP will first do the tasks of GEMM, and when it passes to the tasks of TRSM, a large number of loading will be requested. It should be noted that DMDAR achieves this order because the tasks arrive in the natural order which induces a good POTRF-TRSM-GEMM sequence.

Performances of HFP HFP alternate SYRK, GEMMs and TRSM tasks. GEMM share task with TRSM and SYRK, so this allow a good data re-use. The average improvement of HFP with U order and Belady over the other heuristics is as follows:

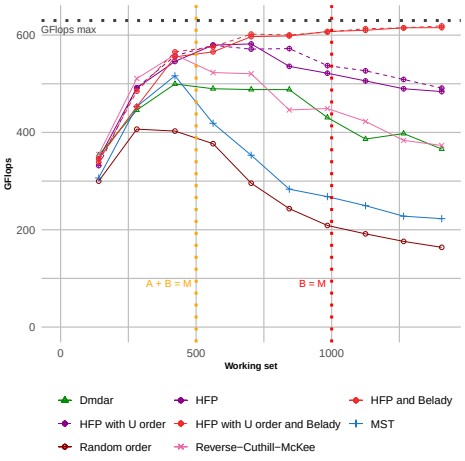
Algorithms	Random	MST	RCM	DMDAR	HFP	HFPU	HFP + belady
Improvement	127.1%	60.4%	57.1%	6.2%	17.2%	13.2%	1.5%

Varying the size of the memory As we saw on figure 7a, we can see on figure 7b that once all the data fit, 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.

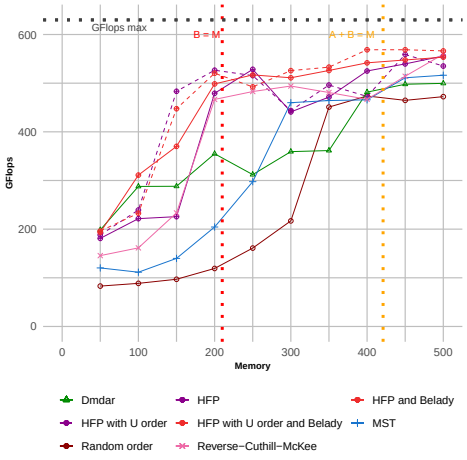
Data transfers Again we can see on figure 7c that the number of data transfers are equivalent for HFP Belady and Dmdar whereas the number of GFlops of HFP with Belady are greater. Again the distribution of loads over time is better for HFP.

4.5 Results on the randomized 2D matrix multiplication

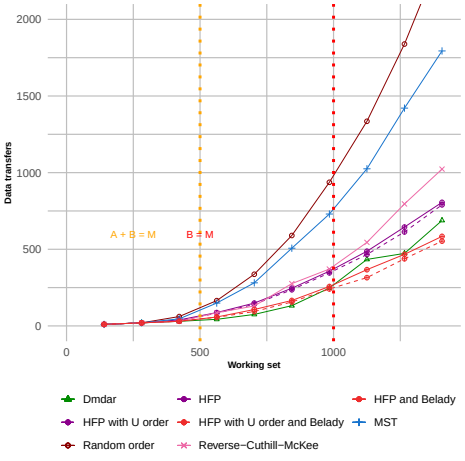
The randomization of data dependencies of the 2D matrix multiplication has an interesting impact on the results previously discussed from Figure 4a. Figure **??**



(a) Varying the size of the working set



(b) Varying the size of the GPU's memory with a wroking set of 210.9 MB



(c) Varying the size of the working set

Fig. 8: Evolution of GFlops on a random 2D matrix multiplication

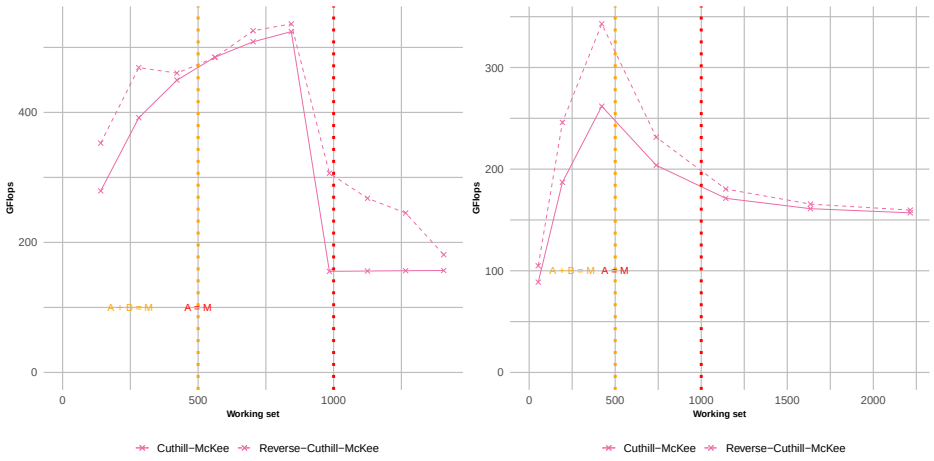
shows that now the MST heuristic does not manage to find task affinities any more. RCM and DMDAR, however, get more sustained performance: the randomization of dependencies actually fades the effect of the classical LRU trap, since they do not tend to execute tasks rows by rows any more. HFP, however, can more nicely find task affinities thanks to its global overview. The addition of the Belady eviction allows to sustain excellent performance, thanks to its insight into the ordering of the randomized tasks.

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

Algorithms	Random	MST	RCM	DMDAR	HFP	HFPU	HFP + belady
Improvement	103.2%	70.3%	23.7%	29.4%	10.6%	8.8%	1.0%

We can also note a bigger difference with or without Belady here (9.5% better with Belady). Indeed, if the data are random, HFP and LRU can no longer rely on the favorable natural order of tasks in 2D. Belady therefore allows improvements in more general cases as well.

4.6 Differences CM and RCM



(a) On a 2D matrix

(b) On cholesky

Fig. 9: Evolution for Cuthill-McKee and Reverse-Cuthill-McKee

We know that the number of loads are the same for an order and it reverse:

Be σ a schedule. We can deduct from σ and the eviction policy chosen (it can be any eviction policy): $\mathbb{S}$ the order in which data are loaded and $\mathbb{V}$ the order in which the data are evicted from the memory. We add all the evictions

for emptying the memory at the end of $\mathbb{V}$. We create the reverse order of σ: $\bar{\sigma}$. We can deduct from $\bar{\sigma}$: $\bar{\mathbb{S}}$ and $\bar{\mathbb{V}}$. We can say that any loading of $\mathbb{S}$ is an eviction of $\bar{\mathbb{V}}$ and any eviction of $\mathbb{V}$ is a loading of $\bar{\mathbb{S}}$. Thus $NB_I(\mathbb{S}, \mathbb{V}) = NB_I(\bar{\mathbb{S}}, \bar{\mathbb{V}})$.

Let's note $minNB_I(\sigma)$ the minimal number of loads we can obtain by using an optimal eviction policy. From $minNB_I(\sigma)$ we can deduct with Belady $\mathbb{S}_{min}$ and $\mathbb{V}_{min}$. Then because Belady is optimal, the number of loads is minimal. We build a couple $(\bar{\mathbb{S}}, \bar{\mathbb{V}})$ that does at most as much loads as the number of loads for $minNB_I(\sigma)$.
$NB_I(\bar{\mathbb{S}}, \bar{\mathbb{V}}) \leq NB_I(\mathbb{S}_{min}, \mathbb{V}_{min}) = minNB_I(\sigma)$
$minNB_I(\bar{\sigma}) \leq NB_I(\bar{\mathbb{S}}, \bar{\mathbb{V}}) \leq minNB_I(\sigma)$ So the reverse order does less loads or as much than the optimal σ.

In the same way we can show that: $minNB_I(\sigma) \leq minNB_I(\bar{\sigma})$

By double inequality: $minNB_I(\sigma) = minNB_I(\bar{\sigma})$.

However the performances are not necessarly the same. We can see on figure 9 that RCM is always better than CM.

In summary, the simple greedy DMDAR heuristic exhibits robust performance, but HFP, and particularly with the U order and Belady variants, showed more sustained performance, including in very restriced situations.

5 Conclusion and Future Work

future work:

- For the moment the tasks are all ready before the start of the execution of the algorithm. A future objective is to manage the hot-start case where tasks arrive in the scheduler little by little. This means that HFP must be able to include tasks in a set of packages.
- adapter aux dépendances
- Adapt our algorithms to multi-gpu problems. Since we are building packages of size M we can distribute these independant packages between GPUs.
- rendement/impact d'un transfert afin d'éviter les pb avec cholesky par exemple

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. 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)

2. 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)
3. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5(2) (1966)
4. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing (Sep 2010), uT-CS-10-659
5. 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)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009), `http://mitpress.mit.edu/books/introduction-algorithms`
7. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th National Conference. ACM '69, Association for Computing Machinery, New York, NY, USA (1969)
8. 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)
9. Gavril: Some np-complete problems on graphs in proceedings of the 11th conference on information sciences and systems. pp. 91–95 (1977)
10. 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)
11. Michaud, P.: (yet another) proof of optimality for min replacement (Oct 2007)
12. 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: Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures. SPAA '13 (2013)