Exploiting data locality to maximize the performance of data-sharing tasksets

Maxime GONTHIER

Inria Bordeaux, 200 avenue de la Vieille Tour 33405 Talence - France
maxime.gonthier@ens-lyon.fr

Abstract
The use of accelerators such as GPUs has become mainstream to achieve high performance on modern computing systems. GPUs come with their own (limited) memory and are connected to the main memory of the machine through a bus (with limited bandwidth). When a computation is started on a GPU, the corresponding data needs to be transferred to the GPU before the computation starts. Such data movements may become a bottleneck for performance, especially when several GPUs have to share the communication bus.
Task-based runtime schedulers have emerged as a convenient and efficient way to use such heterogeneous platforms. With such systems, the scheduler has the ability to choose which task to allocate to which GPU and to reorder tasks so as to minimize data movements. We focus on this problem of *partitioning* and *ordering* tasks that share some of their input data. We present a novel dynamic strategy based on data selection, to efficiently allocate tasks to GPUs, and a custom eviction policy. We compare them to existing strategies using standard scheduling techniques in runtime systems. All strategies have been implemented on top of the StarPU runtime, and we show that our dynamic strategy achieves better performance when scheduling tasks on multiple GPUs with limited memory.

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

1. Introduction

High-performance computing applications, such as simulation for aeronautics, or seismology, keep demanding increasingly intense computation power on ever-growing sets of data. GPUs have risen as an efficient and quick way to run those simulations. They have their own dedicated high-bandwidth memory, which requires transferring data between CPUs and GPUs. To tackle this concern, it has become very common to use the task-based programming paradigm, i.e. to express the application computation as a Directed Acyclic Graph (DAG), and let a dynamic runtime system such as OmpSs [7], PaRSEC [6], or StarPU [2] manage the execution of the task graph with a scheduler.
The memory embedded in GPUs is relatively limited, and the bus that connects them to the main memory has a very limited bandwidth. The runtime scheduler thus not only has to care for task acceleration, it also has to take into account the movement of data within the system. This means that it must carefully decide the task mapping on GPUs according to data locality, as well as the ordering of the tasks itself, to privilege the temporal locality of data, thus favoring data reuse. It is also essential to trigger data transfers ahead of task execution (data prefetches)

so that they can be overlapped. Last but not least, when the embedded memory of the GPU is full, the runtime has to carefully decide which data should be evicted from it, to make room for further data.

We focus in this paper on the problem of **partitioning and scheduling a set of tasks on multiple GPUs with limited memory, where tasks share some of their input data**, as well as managing data movements (loads and evictions). More precisely, we want to determine the order in which tasks must be processed on each GPU to optimize for data reuse as well as maximize overlap between computations and data movements. Our objective is twofold: (i) we partition the work on each GPU to reach a good load balance, and (ii) we want to minimize the total amount of data transferred to the GPUs for the processing of all tasks with a constraint on the memory size. Solving the bi-objective optimization problem for the currently available tasks can lead to a large reduction in data transfers and hence a performance increase, which is ultimately our goal.

In this paper, we make the following contributions:

- We propose a new heuristic based on performing as much computations as possible with data at hand, as well as an eviction policy focused on finding the least-used data in the future (Section 3.2).
- We implement it into the StarPU runtime and first study the performance obtained on an independant set of task: the 2D-blocked matrix multiplication
- We extend our scheduler to deal with dependencies and use the Cholesky decomposition to evaluate it.

Overall, our evaluation shows that our heuristic surpasses previous strategies, in particular in the most constrained situations.

2. Problem modeling

We consider the problem of scheduling tasks on K GPUs, denoted by $\text{GPU}_1, \dots, \text{GPU}_K$. As proposed in previous work [9], 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 $\mathbb{T} = \{T_1, \dots, T_m\}$ and on the other side the data $\mathbb{D} = \{D_1, \dots, D_n\}$. An edge connects a task T_i and a data D_j if task T_i requires D_j as input data. For the sake of simplicity, we denote by $\mathcal{D}(T_i) = \{D_j \text{ s.t. } (T_i, D_j) \in E\}$ the set of input data for task T_i. Each data D_j takes in memory a size w_j. Each of the K GPUs is equipped with a memory of limited size M. Initially, all input data are stored in the main memory of the machine. During the processing of a task T_i on GPU_k, all its inputs $\mathcal{D}(T_i)$ must be in the memory of GPU_k. Note that we do not consider the data output of tasks.

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

3. Algorithms

3.1. Dynamic scheduler of StarPU: DMDAR

"Deque Model Data Aware" (see Algorithm 1 in the complementary material) is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing units in the StarPU runtime[1]. It takes data transfer time into account and schedules tasks where their completion

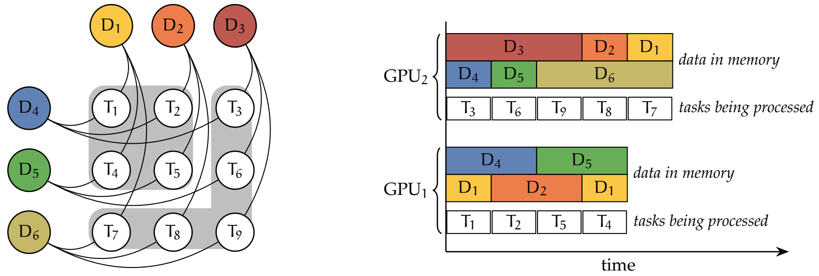


Figure 1: Small example with a 2D grid dependencies. The graph of input data dependencies is shown on the left, together with the task partition among GPUs. A possible schedule is described on the right. Each GPU memory is limited to $M = 2$ and each data has the same size: 1. GPU$_1$ processes tasks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

times (data transfer included) is expected to be minimal. It computes the expected completion time $C_k(T_i)$ of the first task T_i in the queue on each GPU$_k$, based on a prediction of the time for transferring the data to the GPU (or communication time) *comm* and of the task computation time *comp*:

$$C_k(T_i) = \sum_{\substack{j \in \mathcal{D}(T_i) \\ D_k \notin \mathit{InMem}(k)}} \mathit{comm}_k(D_j) + \mathit{comp}_k(T_i) \tag{1}$$

The data transfer time is counted only if the data is not already in the memory of GPU$_k$. Then, the task is allocated on the GPU where its completion time is minimal. Here, we consider the DMDAR variant, which includes an additional *Ready* strategy: on each GPU, tasks are reordered at runtime in order to favor tasks with the most input data already loaded into memory.

3.2. Data-Aware Reactive Task Scheduling

We propose here a novel dynamic strategy, called DARTS (for Data-Aware Reactive Task Scheduling), adapted from previous algorithms specifically designed for linear algebra operations, such as outer products and matrix products [3]. The paradigm is to to perform as many tasks as possible with the data at hand.

The main idea of our new algorithm, detailed in Algorithm 2, is to first consider data movement before task allocation. Whenever some GPU$_k$ requests some new task, we first look in the set *dataNotInMem*$_k$ (which initially contains all data of available task) for the data D that, if moved into the memory of GPU$_k$, would maximize the number of new "free" tasks, i.e., tasks that can be allocated and processed on GPU$_k$ without any additional data movement. Once such a data is found, all these tasks are allocated on GPU$_k$. More precisely, they are put in a local data-structure (*plannedTasks*$_k$) from which tasks are popped one after the other upon request. The process is started again when *plannedTasks*$_k$ is empty. It may happen that we do not find any data that enables some free task. It occurs for example at the very beginning of the computation when all tasks depend on two or more data: at least two data must be loaded in order to produce some free task. In this case, some random unprocessed task is allocated to GPU$_k$. On the contrary, when there exists several candidate data which may produce the maximum number of "free" tasks, we select a data among the candidates that is useful for the highest number of tasks and has the highest priority.

We also designed a custom eviction policy: since we plan ahead which tasks will be allocated to a GPU (through the use of *plannedTasks*), we can take this information into account when we have to remove some data from the memory. This strategy is named Least Used in the Future (LUF) and is detailed in Algorithm 3. We consider all data currently in memory and check if they are used as input for a future task. There are two types of such future tasks: tasks in $\textit{plannedTasks}_k$ that have been reserved for a later allocation on the GPU as mentioned above, but also tasks in $\textit{taskBuffer}_k$, which are the tasks that have been popped from $\textit{plannedTasks}_k$ for execution on GPU_k, and thus whose GPU placement cannot be changed any more (the required data for these tasks may already have been prefetched). We first try to evict a data which is not useful for any task in $\textit{taskBuffer}_k$, and which is an input of few tasks in $\textit{plannedTasks}_k$. If this is not possible, we apply Belady's rule [4] on tasks already allocated: we select the input data whose next usage in $\textit{taskBuffer}_k$ is the furthest in the future, which is known to minimize data movement.

4. Experimental results

In the following experiments we measure the obtained performance as the throughput of elementary computational operations performed per time unit (in GFlop/s) as well as the total volume of data transferred between CPU and GPUs (which we try to minimize). Our goal is to show that through a reduction in data transfers we achieve a better throughput.

4.1. Settings

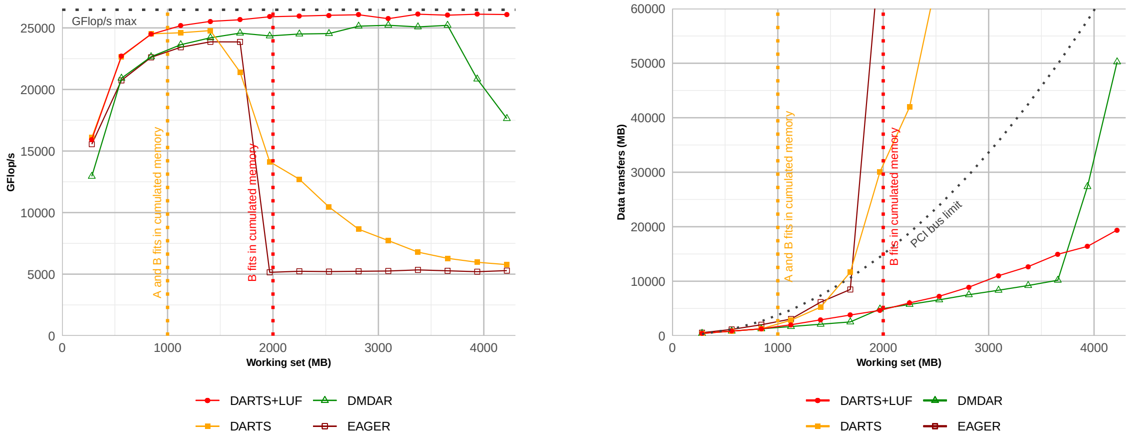
All strategies mentioned above have been implemented in the StarPU runtime system [2]. We performed real experiments on tesla V100 GPUs. We have limited the GPUs memory (500 MB for the outer product and 2000 MB for Cholesky) in order to better distinguish the performance of different strategies even on small datasets. Our main application scenario are tasks from a 2D matrix multiplication with tiles of size 960×960: the matrix product $C = A \times B$ is decomposed into tasks corresponding to the multiplication of one block-row of A with one block-column of B. Input data is thus the rows of A and columns of B. Our second application is the Cholesky factorization with tiles of size 1920×1920 in single precision.

We use the two scheduling heuristics presented above, together with the baseline EAGER scheduler that lets GPUs pick up tasks on demand from a shared queue that contains the tasks in the natural order. All the schedulers use the LRU eviction policy except for DARTS+LUF. When measuring GFlop/s, the cost of computing the schedule is considered. Each result is the average of the performance obtained over 10 iterations. For most of the results, the deviance is less than 2%, thus, we do not show error bars in the following graphs.

4.2. Result on the Matrix Multiplication

General overview

On Figure 2a the dotted horizontal black lines represents the maximum performance that the GPU can achieve when processing elementary matrix products (without I/Os) and is thus our asymptotic goal. On Figure 2a and 2b the red dotted vertical line denotes the situation when the GPUs *cumulated* 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. Figure 2b shows the amount of data transfers. There, the black dotted curve represents the maximum number of transfers that can be done during the minimum time for computation (given by the bound on the throughput), thus the hard limitation induced by the PCI bus bandwidth: a strategy exceeding this amount necessarily requires more time for the data transfers than the optimal



(a) GFlop/s.

(b) Amount of data transfers.

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

time for computation.

EAGER's results

The EAGER heuristic switches to a pathological behavior at the red line. We can see the throughput plummeting (Figure 2a) . It is accompanied with a rise in data transfers after the red line as you can see on Figure 2b. EAGER tends to process tasks along the rows of C. This allows to reuse the same block-row of matrix A for tasks that compute tiles of the same row of C, but requires reloading the whole matrix B for each new block-row of A when the memory is constrained, which is a well-known pathological case of the LRU eviction policy. This explains why EAGER's performance drops at the red dotted line.

DMDAR's results

DMDAR does not suffer from the pathological case affecting EAGER because its *Ready* strategy allows it to rather process tasks that need the block-column of B already in memory instead of reloading the whole B matrix. DMDAR's data transfers however start to rise for the last two working set sizes as we can see on Figure 2b. That corresponds to the performance drop of the last two points of Figure 2a. This is due to a conflict between data prefetching and eviction. Indeed, once the GPU is filled with data, it is not clear for DMDAR whether some data should be evicted in order to perform more prefetches. It will thus rather stop prefetching data as long as all the data currently in the GPU will be useful for the subsequent tasks to be executed there. Also, when some data is actually evicted from a GPU, DMDAR does not reconsider the task mapping according to the new set of data loaded on the GPU. Thus, it cannot make a balance between prefetching and eviction.

DARTS' results

On Figure 2a, DARTS+LUF achieves near perfect performance and has a 9.4% improvement over DMDAR. Indeed, loading a single data that enables multiple tasks minimizes data transfers and increases overlap between transfers and computations. However, after the orange-dotted line, when the memory is constrained, DARTS without the custom LUF eviction policy has to load and evict data following LRU's policy. So the tasks in *taskBuffer* that are supposed to be ready for computation, have to load more data. Indeed, the previous tasks caused evictions, and the data evicted might be needed by the tasks in *taskBuffer*. This causes a domino effect where each new task requires a new data load. For DARTS+LUF, when an eviction is needed, it avoids as much as possible evicting a data that is used by the few tasks already planned for computation. This allows us to avoid the pathological case of DARTS, and to achieve a better balance between prefetching and eviction since DARTS+LUF maintains in *plannedTasks* and *taskBuffer* an accurate overview of the tasks to be computed, even when eviction removes some data from a GPU.

To deal with multiple GPUs, DARTS assigns to each GPU its own set of data $dataNotInMem_k$ to pick from. However all GPUs share the same set of tasks. Once a task is allocated to a GPU, it is removed from the common set of available tasks. Thus our scheduler will naturally assign to the other GPUs data from a line or row that has not been used for tasks yet. This will evenly distribute tasks among GPUs and mostly separate data usage between GPUs.

4.3. Result on the Cholesky factorization

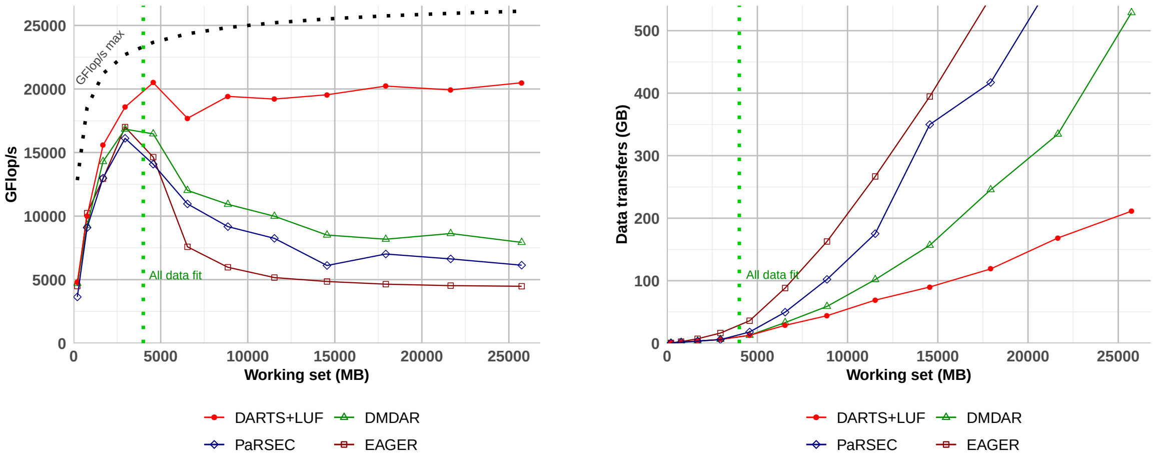
To tackle the issue of a taskset with dependencies, DARTS was first adapted to deal with a dynamic submission of task. $dataNotInMem_k$ is now a list that is updated upon the arrival of new unprocessed data. Moreover, when choosing the optimal data to load D_{opt}, DARTS now looks at the maximum priority (computed from the bottom-level [8] i.e. the time to completion if the platform is not constrained) of the tasks associated with the candidate data. This priority is used as a tiebreak after the number of "free" task that D_{opt} enables.

Figure 3 shows the results on the Cholesky decomposition with 2 GPUs. Here, the green vertical line marks the working set size where all of the input data fit into memory. In this experiment, we compare our solution DARTS to DPLASMA, an implementation of dense linear algebra applications using the state of the art PaRSEC runtime [5].

Apart from DARTS, we observe that all the schedulers suffer from the memory constraint, associated with a rise in data transfers on Figure 3b. By looking at the priorities and the amount of work associated with a data , DARTS is able to have a healthy balance between critical path progression and locality. If only a few tasks are ready, the priority will guide the choice of D_{opt} toward task on the critical path. On the contrary, if a lot of tasks are ready, one can more easily find locality and thus a large number of tasks will be computed with the same data. Moreover, DARTS requires significantly less data transfers than it's competitor as we can see on Figure 3b. This reduction is associated with better performance.

5. Conclusion

Limiting data movements is crucial for performance in modern computing platforms, and especially for machines equipped with several GPUs sharing the same communication bus to the main memory. We proposed a new strategy, named DARTS, which achieves very good performance when the memory is a scarce resource through a reduction in data transfers and a better overlap of communications and computations. DARTS fills each GPU with as many task as possible when they are idling. When you have a large number of GPUs, letting a GPU idling could be a good solution as it enables other GPUs to maximize their data locality. Tackling this issue could allow us to scale DARTS on larger machines. In efforts to make DARTS more generic, we would like to adapt it to heterogeneous platforms by using both CPUs and GPUs. Lastly, we would like to experiment using out-of-core applications, since DARTS is not specific to GPUs and could be applied to any problem where tasks are sharing input data.



(a) GFlop/s.

(b) Amount of data transfers.

Figure 3: Performance on the Cholesky decomposition with 2 Tesla V100 GPUs.

6. Complementary material

Algorithm 1 Deque Model Data Aware heuristic (DMDA)

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

Algorithm 2 DARTS on GPU_k

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

Algorithm 3 Eviction procedure LUF for DARTS on GPU_k

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

Bibliographie

1. Augonnet (C.), Clet-Ortega (J.), Thibault (S.) et Namyst (R.). – Data-Aware Task Scheduling on Multi-Accelerator based Platforms. – In *Int. Conf. on Parallel and Distributed Systems*, décembre 2010.
2. Augonnet (C.), Thibault (S.), Namyst (R.) et 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*, vol. 23, 2011.
3. Beaumont (O.) et Marchal (L.). – Analysis of dynamic scheduling strategies for matrix multiplication on heterogeneous platforms. – In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*, pp. 141–152. ACM, 2014.
4. Belady (L. A.). – A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, vol. 5, n2, 1966.
5. Bosilca (G.), Bouteiller (A.), Danalis (A.), Faverge (M.), Haidar (A.), Herault (T.), Kurzak (J.), Langou (J.), Lemariner (P.), Ltaeif (H.), Luszczek (P.), YarKhan (A.) et Dongarra (J.). – Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. – pp. 1432–1441, Anchorage, Alaska, USA, 2011-05 2011. IEEE.
6. Bosilca (G.), Bouteiller (A.), Danalis (A.), Faverge (M.), Hérault (T.) et Dongarra (J.). – PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering*, vol. 15, n6, novembre 2013.
7. Bueno (J.), Planas (J.), Duran (A.), Badia (R. M.), Martorell (X.), Ayguade (E.) et Labarta (J.). – Productive programming of GPU clusters with OmpSs. – In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 557–568. IEEE, 2012.
8. Casanova (H.), Legrand (A.) et Robert (Y.). – *Parallel Algorithms*. – CRC Press, 2008.
9. Kaya (K.), Uçar (B.) et Aykanat (C.). – Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories. *J. Parallel Distributed Comput.*, vol. 67, n3, 2007.