Composing multiple StarPU applications over heterogeneous machines: A supervised approach

Andra Hugo, Abdou Guermouche, Pierre-André Wacrenier, Raymond Namyst

Noname manuscript No.
(will be inserted by the editor)

Composing multiple StarPU applications over heterogeneous machines: a supervised approach

A.-E Hugo · A. Guermouche · P.-A. Wacrenier · R. Namyst

Abstract Enabling HPC applications to perform efficiently when invoking multiple parallel libraries simultaneously is a great challenge. Even if a uniform runtime system is used underneath, scheduling tasks or threads coming from different libraries over the same set of hardware resources introduces many issues, such as resource oversubscription, undesirable cache flushes or memory bus contention.

This paper presents an extension of StarPU, a runtime system specifically designed for heterogeneous architectures, that allows multiple parallel codes to run concurrently with minimal interference. Such parallel codes run within *scheduling contexts* that provide confined execution environments which can be used to partition computing resources. Scheduling contexts can be dynamically resized to optimize the allocation of computing resources among concurrently running libraries. We introduce a *hypervisor* that automatically expands or shrinks contexts using feedback from the runtime system (e.g. resource utilization). We demonstrate the relevance of our approach using benchmarks invoking multiple high performance linear algebra kernels simultaneously on top of heterogeneous multicore machines. We show that our mechanism can dramatically improve the overall application run time (-34%), most notably by reducing the average cache miss ratio (-50%).

1 Introduction

Due to recent evolution of High Performance Computing architectures toward massively parallel heterogeneous multicore machines, many research efforts have recently been devoted to the design of runtime systems able to provide programmers with portable techniques and tools to exploit such complex hardware. The availability of mature implementation of such runtime systems (e.g. Cilk [14], OpenMP or Intel TBB [11] for multicore machines, Anthill [28], DAGuE/Parsec [8], Charm++ [19], Harmony [12], KAAPI [18], Qilin [20], StarPU [6] or StarSs [7] for heterogeneous configurations) has allowed programmers to rely on thread/task facilities to develop efficient implementations of parallel libraries (e.g. Intel MKL [10], FFTW [13]). The MAGMA library [30], that provides Linear Algebra algorithms over heterogeneous hardware and that uses the StarPU runtime system to perform dynamic scheduling between CPUs and GPUs, well illustrates this trend toward delegating

A.-E Hugo · A. Guermouche · P.-A. Wacrenier · R. Namyst
Inria, LaBRI, University of Bordeaux

scheduling to the underlying runtime system. Moreover, such libraries often exhibit state-of-the-art performance, resulting from heavy tuning and strong optimization efforts.

Consequently, building high performance computing applications on top of such specific parallel libraries is now commonplace [15]. However, even if a natural approach would be to rely on as many external parallel libraries as needed and allow their concurrent execution, most applications invoke only one parallel library at a time. The reason lies in current implementations of parallel libraries not being ready to run simultaneously over the same hardware resources. This problem is referred to as the parallel *composability* problem [22, 21]. This situation is actually alarming, because it reveals that well-known programming principles such as code *composability* and code *reusability* are currently not applicable to High Performance Computing.

There is a wide panel of applications that face this problem, ranging from code-coupling applications (e.g. molecular dynamics coupled with finite elements methods), where opportunities for executing concurrent parallel kernels are still under-exploited, to linear algebra libraries, and more precisely sparse linear algebra methods and fast multipole methods. Typically, numerical factorizations of sparse matrices involve the execution of various dense linear algebra kernels. Some of these kernels operate on small and medium blocks, and thus exhibit poor scalability on high numbers of cores. In such situations, running several kernels concurrently to preserve good scalability of each instance may greatly help to improve overall performance.

Indeed most runtime systems impose restrictions regarding how different parallel constructions that can be mixed and exhibit severe performance issues when trying to simultaneously run independent parallel blocks within the same application. To fully tap into the potential of *many-core* architectures, parallel libraries typically allocate and bind one thread per core to bypass the underlying operating system's scheduler. Specialized parallel libraries, such as Basic Linear Algebra Subroutines (BLAS) for instance, strictly follow such a rigid approach, to better control cache utilization. As a result, applications resulting from the composition of parallel libraries usually exhibit poor performance, because each library is unaware of other libraries' resource utilization. This issue has led several runtime system designers to provide implementations able to avoid *resource oversubscription* when multiple libraries simultaneously request the scheduling of tasks/threads [11].

Moreover, some advanced environments allow for partitioning hardware resources, following an approach similar to virtual machines [22]. The main challenge actually consists in addressing the problem of automatically adjusting the amount of resources allocated to each partition.

In this paper, we present a runtime system architecture where multiple parallel libraries run in different *scheduling contexts* that can be dynamically resized. A scheduling context encapsulates an instance of the runtime system, and runs on top of a subset of the available processing units (i.e. regular cores or GPU accelerators). Contexts allow different libraries to run with limited interference over the same machine. In order to maximize the overall efficiency of applications, contexts can be dynamically shrunk or expanded by a *hypervisor* that periodically gathers performance statistics inside each context (e.g. resource utilization, computation progress) and tries to determine how resources should be assigned to contexts so as to minimize the overall execution time.

We have implemented our approach as an extension to the StarPU runtime system designed for heterogeneous machines [6]. Existing linear algebra applications kernels developed on top of StarPU transparently benefit from the isolation capabilities of contexts with no modification to the original code. We show that our approach leads to significant performance gains compared to situations where parallel code arbitrarily mix over the whole pool

of processing units, and to situations where parallel codes execution is confined inside static contexts.

The contributions of this paper are as follows:

- We describe the composability problem and show that parallel codes can behave poorly when causing mutual interferences
- We design and implement an extension to the StarPU runtime system capable of isolating parallel codes into scheduling contexts
- We implement a hypervisor capable of dynamically resizing scheduling contexts based on performance statistics and resource utilization
- We present performance results that show how our solution proves great potential in improving the behavior of applications relying on several parallel codes/libraries

2 StarPU and the composition problem

2.1 The StarPU Runtime System

The StarPU runtime system [6] is a C library that provides programmers with a portable interface for scheduling dynamic graphs of tasks onto a heterogeneous set of processing units (*i.e.* CPUs and GPUs). The two basic principles of StarPU are firstly that tasks can have several implementations, for some or each of the various heterogeneous processing units available in the machine, and secondly that necessary data transfers to these processing units are handled transparently by StarPU. StarPU tasks are defined as multi-version kernels, gathering the different implementations available for CPUs and GPUs, associated to a set of input/output data. To avoid unnecessary data transfers, StarPU allows multiple copies of the same registered data to reside at the same time on several processing units as long as it is not modified. Asynchronous data prefetching is also used to hide memory latencies.

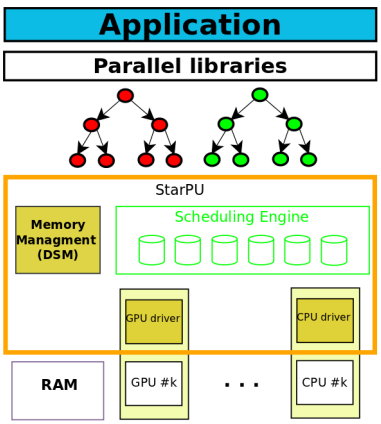


Fig. 1 The architecture of the StarPU runtime system.

2.2 The StarPU Scheduler

StarPU is a platform for developing, tuning and experimenting with various task scheduling policies in a portable way (see Figure 1). Implementing a scheduler consists in creating a set of queues, associating them with the different processing units, and defining the code that will be triggered each time a new task gets ready to be executed, or each time a processing unit is about to go idle. Various designs can be used to implement queues (e.g. FIFOs or stacks), and they can be organized according to different topologies. Several built-in schedulers are available, ranging from greedy and work-stealing based policies to more elaborate schedulers implementing variants of the Minimum Completion Time (MCT) policy [31]. This latter family of schedulers builds on auto-tuned history-based performance models that provide estimations of the expected durations of tasks and data transfers.

2.3 Composability-related issues

Parallel kernels often have specific requirements in terms of scheduling, therefore different algorithms of scheduling are used in order to satisfy the granularity of the parallelism and to deal with dependencies between tasks. However when composing different parallel kernels this issue rises and it makes the composition of different schedulers, usually incompatible one with another, difficult. Thus, a first step on the path to composability is to be able to deal with multiple-schedulers.

The second issue is that StarPU does not provide kernel isolation mechanisms. StarPU, like most task-based runtime systems, uses an online scheduling policy to assign the tasks submitted by the application to the various processing units. When confronted to simultaneous task flows, these online scheduling techniques may fail to deal with resource sharing, resulting in a deterioration of data locality and scheduling quality.

Indeed, kernel programmers often tune their codes to force runtime scheduler decisions by taking task submission order into account, pre-allocating memory attached to a specific device or by introducing priorities. For instance, tasks along the critical path are often given high priorities. Such hints cannot be inferred automatically by a runtime system.

Interleaving unrelated tasks over a same device may ruin such optimizations: when running several unrelated parallel codes on top of StarPU, tasks coming from different codes will mix and compete for resources in a way that cannot be controlled anymore by the programmer.

2.4 Discussion

Composing multiple StarPU parallel codes efficiently while limiting their mutual interference could theoretically be seen as a global scheduling problem. Indeed, multiple parallel kernels relying on different schedulers could simply been merged, provided that a *super-scheduler* could meet the requirements of each individual kernel. This problem is related to the co-scheduling of multiple parallel jobs which share the underlying processing units. This has been extensively studied for cache-sharing in multi-processor platforms. Theoretical studies show that the problem is NP-Complete and can be solved only for very simple architectures [29]. Moreover, scheduling policies may be so diverse that the optimization criteria would be different (e.g. time to completion, power consumption, etc.) In such situ-

ations, there would simply be no rationale that would help the super-scheduler to prioritize tasks coming from different parallel kernels.

Thus, such a super-scheduler would have no other choice but to allocate separate resources to each parallel code. It would also have to dynamically adapt to new incoming kernels and their associated scheduling policies, and hence would probably have to perform a dynamic resource allocation between kernels. Such a super-scheduler would suffer from a scalability problem though, since it would have to maintain a global view of the whole set of computing resources despite the fact that each parallel kernel would only use a subset of them.

3 Our Approach to co-scheduling multiple parallel codes

In this section, we describe our generic approach to tackle the issues described in previous sections. Instead of trying to design a super-scheduler as discussed previously, we propose a solution where we split the resources into sets managed by different scheduling algorithms. This is done through the introduction of the so-called *Scheduling Contexts* which are abstract sets of resources that allow programmers to control the distribution of computational resources (*i.e.* CPUs and GPUs) to concurrent parallel codes. The main goal is to minimize interferences between the execution of multiple parallel kernels, by partitioning the underlying pool of resources using contexts. Such a property is critical for high performance parallel kernels which are very sensitive to data locality within caches (*e.g.* level 3 BLAS routines) and embedded memories of the GPUs. Indeed ignoring data locality when taking scheduling decisions results in serious memory contention issues, and puts scalability at stake. Moreover since there does not exist a single perfect scheduling strategy that would be suitable for every parallel kernel library, each scheduling context encapsulates its own scheduler that has only a limited view of hardware resources.

Similarly to lightweight virtual machines, *Scheduling Contexts* allow a flexible partition of the machine and unmodified parallel kernels to coexist. StarPU schedulers run unmodified as guest schedulers in an isolated manner.

We have implemented a Scheduling Context layer within StarPU runtime system in order to study their behavior on heterogeneous machines. StarPU is a runtime system that tightly integrates data management and scheduling support. It proposes a unified abstraction of different processing units, which allows us to easily manipulate resources between and inside the contexts.

We place the Scheduling Context layer above the Scheduling Engine of StarPU, without actually interfering with the implementation of the schedulers (Figure 2). By using a black box approach, the scheduler receives information regarding the processing units it should execute on and returns a valuable distribution of tasks over the restrained group of resources. Thus, the main challenge is to distribute computing resources to schedulers so that they better meet the constraints of the application. The scheduling policies may be very diverse: they can aims at minimizing the termination time of the kernel, minimizing the memory occupation or maximizing the efficiency of the processing units.

To this end, we introduce a dynamic approach to resize the contexts. This is mainly motivated by the fact that it is not always easy or even possible to define a good distribution of the resources among contexts *a priori* (neither statically at the compile time, nor at the beginning of the execution of a kernel). Indeed, the application may be irregular and therefore the requirements of its kernels in term of resources may change during their execution. Moreover, it may be more convenient for the programmer to specify coarse grain bounds

on the number of resources belonging to each context and letting the runtime system refine its distribution dynamically. This approach represents a good combination of the high-level expertise of the programmer and the low-level view of the runtime system to ensure portability of performance. To do so, we introduce a hypervisor which is in charge of managing the resources allocated to each context in a dynamic way using different metrics coming from both the application and the StarPU runtime system.

3.1 Extending StarPU with scheduling contexts

3.1.1 Architecture

To allow multiple StarPU kernels to run concurrently while keeping the scheduling policies simple and effective, we propose to run the different parallel kernels (task based computations in the case of StarPU) separately by isolating them into *scheduling contexts* (Figure 2). Kernels are executed in a confined way so as to improve data locality, lower memory contention and increase performance of the whole application.

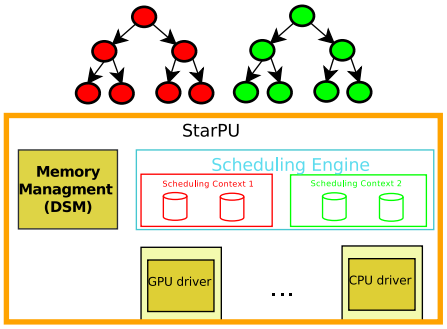


Fig. 2 Scheduling Contexts in StarPU.

Each scheduling context is associated with a scheduling policy, which allows several schedulers to coexist with limited interference within a single parallel application. Most importantly, a scheduling context can have a restricted view of the hardware: a list of "visible" processing units (regular cores, accelerators, etc.) is maintained for each context by the runtime system. Contexts thus represent a convenient tool for partitioning the set of available processing units.

Note that some specific types of processing units, such as GPUs, can not always be exploited at their full potential by some kernels. This is mainly due to the fact that a given parallel kernel may not have enough tasks capable of running on such accelerators. To tackle this problem, our mechanism allows any resource to be time-shared between several contexts. When a processing unit is shared by several contexts, StarPU uses a round-robin algorithm between the different contexts in order to fetch the next task to run.

Since computing resources may be shared by multiple contexts, the associated task schedulers need to cope with processing units interleaving tasks coming from multiple contexts. We have thus modified the schedulers provided in StarPU in order to be able to correctly predict the expected termination time for the resources shared between contexts. This

is done by making the contexts inform each other when they schedule tasks on these resources. Thus, each scheduler associated to a context is aware of all the tasks assigned to the shared resources, even the ones coming from other contexts.

3.1.2 Execution model

Scheduling contexts can be created or destroyed dynamically, as libraries or kernels are not necessarily initialized at the same time and they may not be used during the entire application. When creating a context, the programmer indicates the processing units and the scheduling policy to be used for executing parallel kernels (see Figure 3).

He also has to specify to which of the previously created contexts he wants to submit tasks.

```
int resources1[3] = {CPU_1, CPU_2, GPU_1};
int resources2[4] = {CPU_3, CPU_4, CPU_5, CPU_6};

/* define the scheduling policy and the table
  of resource ids */

sched_ctx1 = starpu_create_sched_ctx("heft",resources1,3);

sched_ctx2 = starpu_create_sched_ctx("greedy",resources2,4);
```

```
// thread 1:

/* define the context associated to kernel 1 */
starpu_set_sched_ctx(sched_ctx1);

/* submit the set of tasks of the parallel kernel 1*/
for( i = 0; i < ntasks1; i++)
 starpu_task_submit(tasks1[i]);
```

```
// thread 2:

/* define the context associated to kernel 2 */
starpu_set_sched_ctx(sched_ctx2);

/* submit the set of tasks of parallel kernel 2*/
for( i = 0; i < ntasks2; i++)
 starpu_task_submit(tasks2[i]);
```

Fig. 3 Programming with Scheduling Contexts

3.1.3 Allocation of processing units

It is worth to note that high performance programmers usually know some characteristics of their kernels and have the ability to analyze and understand the performance of their application. Thus, it is crucial to let them specify how resources should – roughly or precisely – be distributed among the contexts. To this end, we give the programmer a way to define a specific distribution.

If the programmer does not provide this information, we propose two algorithms to compute an estimated distribution of resources over the contexts depending on the amount

of work (that is, the number of floating point operations or the tasks) associated with each context. The first algorithm involves the resolution of the linear programming problem described by Equation (1) where we compute the number of CPUs and GPUs needed by each context such that the program will end its execution in a minimal amount of time. Note that this is a rough approximation since we do not consider either task dependencies or task specificities.

$$\max\left(\frac{1}{t_{max}}\right) \text{subject to} \left(\begin{array}{l} \left(\forall c \in C, n_{\alpha,c}v_\alpha + n_{\beta,c}v_\beta \geq \frac{W_c}{t_{max}}\right) \\ \wedge \left(\sum_{c\in C} n_{\alpha,c} = n_\alpha\right) \\ \wedge \left(\sum_{c\in C} n_{\beta,c} = n_\beta\right) \\ \wedge \left(\forall c \in C, n_{\alpha,c} < max_{\alpha,c}\right) \\ \wedge \left(\forall c \in C, n_{\beta,c} < max_{\beta,c}\right) \end{array}\right) \quad (1)$$

In this linear program C denotes the set of contexts, $n_{\alpha,c}$ and $n_{\beta,c}$ represent the unknowns of the system, that is the number of CPUs and GPUs that are assigned to a context c, W_c is the total amount of work associated to the context c, t_{max} represents the maximum amount of time spent by a context to process its amount of work, v_α and v_β represent the speed (i.e. floating point operations per second) of a CPU respectively GPU on the platform, n_α and n_β are the total number of CPUs, respectively GPUs available on the machine. Equation (1) expresses that each context should have the appropriate number of CPUs and GPUs such that it should have finished its assigned amount of work before the deadline t_{max}. Of course, this linear program can be easily generalized to platforms with more than two types of resources.

For the second algorithm, we focus on a more accurate information concerning the workload of the application. We consider task requirements (affinity with computational resources, execution time). The estimated execution time of each type of task on each processing unit is provided by the StarPU performance model system. By recording the number of submitted tasks according to their type (the kernel they run, the size of data they operate on and the context they belong to) we can compute an optimal distribution of tasks among processing units. The main improvement of this algorithm is that it takes into account the type of tasks and does not consider the application as a simple amount of work to be done. The problem corresponding to the resources allocation in this case is solved through a non-linear program where we want to minimize the global completion time. To solve this non-linear program, we use a dichotomy on the value of the variable causing the non-linearity (t_{min} in our case) to find its optimal value. At each step of the dichotomy process, we try to find a feasible solution to the linear program described in Equation (2). Note that this linear program does not need a specific objective function since we just need to check if a feasible solution can be computed with the given value of t_{min} (t_{min} being the execution time). Of course, the correctness of this technique relies on the accuracy of the performance models.

$$\min\left(t_{min}\right) \text{ subject to } \left(\begin{array}{l} \left(\forall w \in W, \forall c \in C, \sum\limits_{t \in T_c} n_{t,w} \cdot d_{t,w} \leq t_{min} \cdot x_{w,c}\right) \\ \wedge \left(\forall c \in C, \forall t \in T_c, \sum\limits_{w \in W} n_{t,w} = n_t\right) \\ \wedge \left(\forall w \in W, \forall c \in C, x_{w,c} \in \{0,1\}\right) \\ \wedge \left(\forall w \in W, \sum\limits_{c \in C} x_{w,c} = 1\right) \end{array}\right) \tag{2}$$

In this linear program C denotes the set of contexts, W is the set of workers, T_c represents the set of type of tasks of a context c, $x_{w,c}$ is a boolean variable denoting whether or not a worker w belongs to the context c, $n_{t,w}$ represents the number of tasks of type t performed by the worker w, $d_{t,w}$ is the estimate duration of the execution of the task t on the workers w and finally n_t is the number of tasks of type t encountered during execution. Equation (2) expresses the fact that each worker has to execute its tasks assigned by the context it belongs to before the total execution time t_{min}. Furthermore, each worker has to be part of exactly one context. Finally, it is important to notice that the complexity of this second approach is greater than the one of the first approach (either in term of number of constraints, number of variables, or because of the dichotomy process).

3.2 Dynamic allocation of computational resources

We present in the following section *the hypervisor*, a tool capable of resizing the scheduling contexts whenever their initial configuration deteriorates the performance of the application.

3.2.1 Architecture

The hypervisor is an entity that evaluates the behavior of the processing units inside the contexts and decides whether they should rather be reallocated to other contexts or not. Although our current implementation of the hypervisor is linked to StarPU, it could easily be plugged into another runtime provided it has similar functionalities for adding/removing the processing units to/from contexts as well as profiling capabilities.

The running application is monitored using a set of informations/metrics that are provided either by the application hints or by the runtime system as performance counters (e.g. the time a resource was idle). Hints may typically fix the lower- and upper bounds that the hypervisor should not cross when allocating computing resources to a given context. For example, if the programmer wants to run two different parallel kernels simultaneously within the same application, he provides the hypervisor with a range of processing units that he considers necessary to the execution of each kernel (e.g. at least 1 GPU and between 2 and 4 CPUs for the kernel). Based on this information, the hypervisor adapts the size of the contexts according to its metrics while respecting the constraints given by the programmer.

Thus, the hypervisor blends in a light *Statistics Manager* that stores information about contexts and resources performance. Additionally, a *Resizing Engine* is responsible for redistributing computing resources based on performance prediction of the application.

A small number of low-intrusive callbacks are introduced to trigger the resizing of some contexts when the characteristics of the application no longer match the requirements.

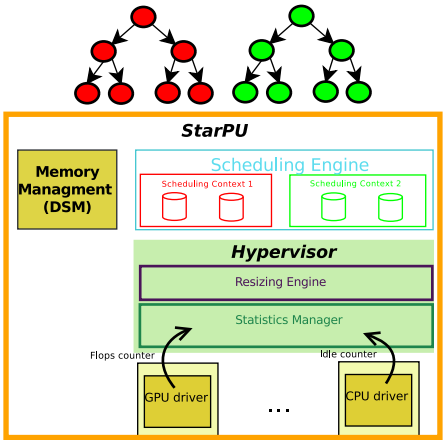


Fig. 4 Placing the Hypervisor in StarPU

3.2.2 Trigger the resizing decisions

The hypervisor (see Figure 4) can either be invoked directly by the application (by creating, destructing or modifying a context during the execution of the application) or be triggered periodically according to the behavior of the application. In this paper, we use a simple metric to trigger the resizing process : *Idleness*, which corresponds to the maximum idle time allowed for a given computing resource. This threshold is defined by the programmer. Alternate metrics, such as application's instant speed (computed by reading hardware counters or by using explicit feedback from the application), could be used as well. The hypervisor decides to resize contexts when the instant speed of kernels running within contexts is below a given threshold. However, for more regular kernels such as Cholesky Factorization, simpler strategies can be used. Indeed, in this case the change in the instant speed of a context is mainly caused by idle time. Therefore in the following section we mainly focus on the Idleness metric and we present the resizing algorithms used whenever idle processing units are detected.

3.2.3 Hypervisor management policies

The philosophy of our redistribution process is that the more information we get from the application, the better the resource distribution. We believe that collaborative strategies mixing knowledge both from the application and from the runtime system form the right way to achieve performance portability. In this respect, workload information is crucial since it allows to *minimize the makespan* of the application. This workload information may be provided either as an average amount of flops or as the number of tasks of each type that will be submitted. It is important to note that even if the application's workload is unpredictable (as it is usually the case with irregular applications), it is usually possible to get a good workload estimation for the current phase of computations (e.g. for a single iteration).

In the situations where the remaining number of tasks of each type is known for each context (e.g. most dense linear algebra algorithms), the hypervisor solves the linear program described by Equation (2). If only an estimation of the speed (e.g. in FLOP/s) of each

completed task can be used, the hypervisor solves the linear program described by the Equation (1). These strategies will be referred to as *completion time based* resizing policies.

However, in the case where no global workload information is available, either because the application is very irregular or because the programmer is not an expert, resizing policies may rely only on runtime feedback. The most relevant strategies tend to optimize a "local" metric : make the instant speed of the different contexts converge to the same value, maximize the throughput of the platform (i.e. assign a resource to the context which uses it in the most efficient way), etc.

So, in order to keep up with the requirements of different parallel applications we provide a framework that allows implementing custom resizing policies. For instance, we have implemented a policy which aims at balancing the instant speed of multiple parallel kernels. In the remainder of the paper, this strategy will be referred to as *instant speed based* resizing policy. Considering that we have no initial guess about the total workload which will be processed by each context, the instant throughput of each context is determined based on computations that have already been completed by the context. The main idea is to divide the execution of the application into several intervals. The hypervisor monitors the performance of the resources in the last interval and tries to adjust the contexts so that they execute at the same instant speed in the next interval. For this purpose we use the non-linear program given by Equation (3).

$$\min\left(t_{min}\right) \text{subject to} \left(\begin{array}{l} \left(\forall w \in W, \forall c \in C, \frac{1}{s_{w,c}} \cdot \theta_{w,c} \leq t_{min} \cdot x_{w,c}\right) \\ \wedge \left(\forall c \in C, \sum_{w \in W} \theta_{w,c} = \theta_c\right) \\ \wedge \left(\forall w \in W, \forall c \in C, x_{w,c} \in \{0,1\}\right) \\ \wedge \left(\forall w \in W, \sum_{c \in C} x_{w,c} = 1\right) \\ \wedge \left(\forall w \in W, \sum_{c \in C} \theta_{w,c} > 0\right) \end{array}\right) \tag{3}$$

In this non-linear program C denotes the set of contexts, W is the set of workers, $x_{w,c}$ is a boolean variable denoting whether or not a worker w belongs to the context c. $s_{w,c}$ is the instant speed of the worker w in the context c, $\theta_{w,c}$ is the number of flops needed to be executed by the worker w in context c out of the total number of flops θ_c. Equation (3) expresses the fact that each worker has to execute the number of flops assigned by the context it belongs to before the total execution time t_{min}. Furthermore, each worker has to be part of exactly one context and to have some flops assigned to it if it is chosen to belong to that context. Finally, it is important to notice that the complexity of this approach is important. As for the Equation (2), the dichotomy process requires the execution of the linear program several times until the good solution is found. Finally, it is important to emphasize that the approach described in this section can be improved if the application provides some high level information about its execution. For instance, if the application is able to provide the proportions of workload assigned to each context, it is straightforward to update Equation (3) to take into account such informations.

```
/* select an existing resizing policy */
struct hypervisor_policy policy;
policy.custom = 0;
policy.name = "idle_policy";

/* initialize the hypervisor and set its resizing policy */
sched_ctx_hypervisor_init(policy);

/* register context 1 to the hypervisor */
sched_ctx_hypervisor_register_ctx(sched_ctx1);

/* register context 2 to the hypervisor */
sched_ctx_hypervisor_register_ctx(sched_ctx2);

/* define the constraints for the resizing */
sched_ctx_hypervisor_ctl(sched_ctx1,
   HYPERVISOR_MIN_CPU_WORKERS, 3,
   HYPERVISOR_MAX_CPU_WORKERS, 7,
   NULL);
```

Fig. 5 Configuration of the hypervisor.

3.2.4 Execution model

We provide in Figure 5 an example illustrating how the programmer can specify constraints to the hypervisor. In order to indicate the minimum and the maximum number of workers allowed in a certain context we can use the function `sched_ctx_hypervisor_ctl`. This way the resizing process is restricted to this interval.

4 Evaluation

In this section, we present a series of experiments which evaluate the impact of using scheduling contexts within applications requiring multiple parallel kernels to be executed concurrently. Typically, sparse linear solvers require several dense linear algebra kernels to be run simultaneously. The same observation can be made about domain decomposition methods where local solvers are called in parallel on each domain. We use benchmarks to study how two (or more) different concurrent kernels will compete for resources and exhibit how our scheduling contexts solve this problem in a generic way.

4.1 Experimental platform

We evaluate the relevance of our approach using the *mirage* platform, a heterogeneous system composed of two Intel hexa-core processors X5650 at 2.67 GHz having 12 MB of L3 cache for a total of 12 cores and 36 GB of main memory, equipped with three NVIDIA Tesla M2070 GPUs having 6 GB of memory each. Note that 3 of 12 cores are devoted to execute NVIDIA GPU drivers.

4.2 Experimental setup

We focus on parallel applications running simultaneously several StarPU-enabled parallel libraries. We perform measurements on parallel kernels from the MAGMA library [30,3].

Such dense linear algebra kernels are characterized by a large number of spawned tasks and by a DAG-shaped dependency graph. We use an implementation of MAGMA based on StarPU [2] which can efficiently exploit hybrid platforms. The amount of tasks depends on both the size of the input matrix and the size of the blocking (tile size) used for the layout of the matrix. We implement simple programs calling multiple instances of MAGMA factorizations simultaneously and we consider the total execution time of the application, because scheduling contexts are expected to improve the overall behavior of the application and not just the performance of each parallel kernel. We selected the Cholesky factorization kernel (`potrf`) for its simplicity and regularity. In the following sections, we will refer to this MAGMA implementation of the kernel when mentioning Cholesky factorizations. Moreover, to ensure best performance for MAGMA kernels, we use two blocking factors for all our experiments, one favorable to GPUs of 960 x 960 elements and one favorable to CPUs of 192 x 192 elements.

On the other hand, we also use the more regular Computational Fluid Dynamic (CFD) benchmark from the Rodinia benchmark suite [9]. This code implements an iterative solver for the three-dimensional Euler equations for compressible fluids. Such a scheme is very representative for unstructured grid problems, which represent an important class of applications in scientific computing. This benchmark has been rewritten to sit on top of StarPU. The parallelization of this solver is done through domain decomposition. The number of tasks is proportional to the number of domains and the number of iterations. The tasks are independent at each iteration while there are dependencies between an iteration and the next one.

In the following sections we present a set of experiments illustrating the benefits of using the scheduling contexts when running MAGMA codes, CFD codes, or both types of codes. We show that scheduling contexts can be configured to better meet the specific needs of each kernel. Certain kernels require more resources than others because they can generate more parallelism and exploit more devices more efficiently. In contrast, some kernels are not able to efficiently exploit certain types of devices. Contexts can be used to run kernels over their favorite subset of resources.

The main library we used for our experiments (c.f. the MAGMA library) is the most efficient (together with FLAME [32, 23]) and used library for dense linear operations on top of heterogeneous systems. Moreover, using task-based runtime systems for designing complex applications is still an ongoing work in different fields. For example, irregular applications like sparse solvers, FMM applications are being studied by the leading groups of each area to adapt them to heterogeneous platforms using these runtime systems. Therefore, it is untimely to evaluate our scheduling contexts using these applications which has not been yet completely ported and validated by their communities. Thus, we decided to illustrate the behavior of the contexts by confronting them to artificial but well-controlled scenarios that mimic the configurations that can be met within these complex applications. Typically, a sparse direct solver features a graph of many dense BLAS operations, many of which can run concurrently.

4.3 Scheduling Contexts

4.3.1 Efficient composition of different parallel kernels

We first explore the benefits of scheduling contexts when mixing two different parallel kernels, having different algorithms and different requirements in terms of parallelism. We

show that by carefully isolating kernels, we can significantly increase the performance of the whole application.

To do so, we execute, on one side the Cholesky factorization of MAGMA library, a parallel kernel very scalable on both CPUs and GPUs and on the other side the CFD benchmark a parallel kernel mostly efficient on GPUs, having strict requirements for the number of GPUs (mostly depending on the partitioning of the underlying mesh). Thus we factorize a matrix of 15 000 x 15 000 elements while executing the CFD solver on 2957K elements throughout 200 iterations. We divide the CFD mesh in two sub-domains and we observe that when running alone, the best performance is obtained with two GPUs (each GPU being associated with a domain). Thus, we present in Table 1, an experiment where the CFD kernel has to be executed together with the MAGMA kernel described above.

	Execution time
Cholesky Factorization and CFD in 1 context	19.83 s
Cholesky Factorization and CFD in 2 contexts	14.26 s

Table 1 Concurrent execution of CFD Benchmark and Cholesky Factorization of the MAGMA library

The scheduling strategy used in this case is aware of the specific needs of CFD and avoids any additional data transfers by restricting the execution of CFD tasks on 2 GPUs. However, the Cholesky Factorization scales very well on all GPUs and CPUs and it obviously interferes, if allowed, with the data locality needed by the CFD on its 2 GPUs. According to the Table 1 isolating the two parallel kernels in different scheduling contexts improves significantly the performance of the overall execution of the application. In this case the Cholesky Factorization is not allowed to execute tasks on the GPUs, preventing data associated to CFD from being flushed from the GPUs' memory. Note that in the case where the contexts are used, 2 GPUs are assigned to CFD and 1 GPU and 9 CPUs to MAGMA.

4.3.2 Enforcing locality

In this section we show that scheduling contexts help to better exploit data reuse and locality. In Table 2, we present an experiment where we execute three independent parallel kernels, performing Cholesky Factorizations on matrices of 20 000 x 20 000 elements (in the case where contexts are used, each kernel is associated to a context). We evaluate statistics concerning the chances of finding the needed piece of data on a certain device memory. We observe that by using the contexts we reach a hit rate of 92 % which is almost 10 % higher than the regular case. Furthermore, if we consider the data transfer statistics we notice that the total amount of data transfered is drastically reduced (more than 50 % reduction) when using the contexts.

4.3.3 Reducing interferences

We next present how applications mixing a greater number of parallel kernels improve significantly their efficiency when isolating the kernels in different scheduling contexts. We illustrate this behavior by executing an application composed of 9 independent Cholesky factorizations of matrices of the same size (20 000 x 20 000 elements), 9 being the number of available CPU workers on our test platform. In the results reported in Table 3 we compare the execution time of the application, when it mixes the kernels into one context to the

	1 context	3 contexts
Hits on Host memory	91.2%	88.8%
Hits on GPU 1 memory	79.1%	93.2%
Hits on GPU 2 memory	78.7%	93.9%
Hits on GPU 3 memory	78.5%	93.7%
Total hits	82.7%	92.2%
Total transfered data (in GB)	27.3	12.7

Table 2 Data transfer statistics of concurrent execution of three factorizations on *mirage* platform

version separating them in several contexts. We have also measured the serial execution of the nine kernels (i.e. the nine parallel kernels are executed one after the other in a single context).

	Total execution time	Total data transfered
1 context : 9 CPUs / 3 GPUs	52.0 s	113 GB
3 contexts : 3 x (3 CPUs / 1 GPU)	34.8 s	37 GB
9 contexts : 9 x (1 CPUs / 0.3 GPU)	34.4 s	41 GB
serial execution	44.3 s	87 GB

Table 3 Concurrent execution of 9 independent Cholesky factorizations of matrices of 20 000 x 20 000 elements

Concurrent parallel kernels isolated in contexts show important performance improvement compared to the mono context version. In our experiment the overall application has reduced its execution time by 34%. The performance degradation of the single context version is coming from GPUs exploitation. We notice an increase of data transfered between the GPUs and the main memory inducing more blocking waits at the GPU side. Thus, all nine kernels try to transfer data to all three GPUs, competing all for the memory. In order to make room for their new data they discard memory areas used by the others, inducing new data transfers.

On the contrary, by separating the nine kernels in three contexts, or even in nine, the number of kernels which use a given GPU is smaller and therefore contention is reduced. To further illustrate this phenomenon we measured the amount of memory transfers between CPUs and GPUs in the three cases. If we consider the misses in the GPUs memory, we observe that when using an appropriate number of scheduling contexts we have around 10% of memory misses while when using a single context version we have 19% of memory misses. Furthermore, the amount of data transfered between CPUs and GPUs is around 37 GB when using several scheduling contexts whereas it reaches 113 GB when using a single context. These measurements illustrate that we have more contention at the GPU level when having a single context which induces more data to be evicted from GPUs and thus more data to be transfered. We reproduced these measurements on larger kernels (i.e. with matrix of order 30 000) and observed roughly the same behavior (multiple context-based configurations are around 30% faster than single context ones) in the sense that without contexts we have more contention on the GPUs which reduces its performance.

We notice that even when executing the nine kernels in a serial way the amount of transfered data is still superior to the one obtained when using the contexts. The main reason is that when executing a kernel on multiple GPU devices, StarPU prefetches data to all of them [5]. Therefore when executing a sequence of kernels, we transfer data, for each one of

them, to the three GPUs. On the contrary, when using contexts the kernels will prefetch data only on the GPU they are allowed to execute on.

It is interesting to notice that separating the kernels in 3 contexts or in 9 contexts does not change the behavior of the application. The reason is that in both cases one GPU is shared by three kernels. In the case of 9 contexts, they overlap over the GPUs and in the case of 3 contexts, we assign one GPU per context, but inside the contexts we have 3 kernels. We noticed then that having a wise management of the GPUs is an important matter and contexts represent a useful tool to do this.

4.4 Using a Hypervisor to dynamically resize contexts

We illustrated in the previous sections the importance of isolating parallel codes into scheduling contexts. We showed that they are a useful tool which allows the programmer to assign the appropriate set of resources to each kernel and improve their efficiency. We can determine the resource distribution over the contexts by doing several experiments or by letting StarPU compute an optimistic distribution. However, statically determining the best distribution is a difficult issue. Indeed, even if an expert can sometimes provide a good initial distribution of resources among contexts, a dynamic mechanism is usually necessary to polish this initial distribution using dynamic redistribution of resources.

In the following sections we present a set of experiments which enlighten these two behaviors and we show that the hypervisor improves the performance of the application by taking decisions of when and what resources to move from one context to another.

4.4.1 Experimental scenarios

In the following experiments we show that the hypervisor can detect an inefficient distribution of resources, find a better one and finally resize the contexts consequently.

We evaluate the behavior of the hypervisor by creating synthetically some negative scenarios determined empirically. In these scenarios we simulate applications arriving at a point in their execution where the distribution of the resources is no longer efficient.

The first application executes concurrently in one context the CFD solver on 2957K cells throughout 200 iterations and in another context a Cholesky Factorization on a matrix of 15 000 x 15 000 elements. By dividing the CFD domain in two sub-domains we observe that the scheduler would distribute the corresponding tasks only on two GPUs in order to avoid unnecessary data transfers. Therefore, in order to disturb the hypervisor and verify if its decisions are correct, we choose in our experiment to assign three GPUs to the CFD kernel and 9 CPUs to the Cholesky Factorization.

The second application is composed of two Cholesky factorizations, one of them executing on a matrix of 15 000 x 15 000 elements using a block size of 192 x 192 elements (CPU friendly) and the second one, on a matrix of 30 000 x 30 000 elements using a block size of 960 x 960 elements (GPU friendly). Therefore, we assign to each context a non-optimal number of processing units and we expect the hypervisor will find a distribution which would be as efficient as the one determined by solving the linear program given by Equation (2).

The optimal distribution gives 9 CPUs to the context corresponding to the factorization of a matrix of 15 000 x 15 000 elements and all the GPUs to the one corresponding to the factorization of a matrix of 30 000 x 30 000 elements (this distribution has been determined by solving the linear program given by Equation (2) and verified empirically). Thus, we give

only 3 CPUs to factorize a matrix of 15 000 x 15 000 elements and all the GPUs and the rest of 6 CPUs to factorize a matrix of 30 000 x 30 000 elements (the so called "Arbitrary distribution").

4.4.2 Magma's Cholesky Factorization and CFD competition for resources

The first scenario, presents an inefficient distribution of resources between the two kernels, the factorization and the CFD. As expected, the factorization would need one GPU while CFD has one on which it does not scale. We see in Table 4 that the intervention of the hypervisor is essential. We use the hypervisor to detect the idle GPU and to resize the contexts and allocate it to the Cholesky Factorization and we notice an important increase of performance.

	Execution time
Arbitrary distribution	53.08 s
Idleness based resized distribution	14.26 s

Table 4 Concurrent execution of CFD Benchmark and Cholesky Factorization of the MAGMA library

CFD requires a good calibration of the performance model of the memory bus [5] as well as good scheduling decisions in order to perform well. If these conditions are met, it scales only on 2 GPUs, leaving the third one idle. Assigning this processing unit to a context which is able to use it decreases significantly the execution time of the overall application.

4.4.3 Impact of the triggering resize criteria

In order to be able to correct the distribution of resources in terms of execution time the resizing decision should be taken at as soon as the application does not perform as expected. In the following section we study the criteria that detects this moment and triggers the resizing process. We consider the first scenario, composing two Cholesky factorization having different workloads and we evaluate the impact of the triggering criterion on the execution of the application.

Resources being about to reach a starvation state in one context may be more efficiently used in another context. Finding the moment at which a worker becomes idle for too long and should no longer belong to a context is actually an easy task. First of all, a resource is considered as idle if it fails n consecutive times to get work. In the rest of the section n will be denoted as the idleness tolerance parameter. Figure 6 illustrates the impact of the idleness tolerance parameter on the global performance of the application. As expected, when the value of this parameter exceeds a threshold, the execution time is negatively impacted. This is mainly due to the fact that resizing decisions are taken too late letting the resources being idle for too long. On the other hand when the threshold of n has a small value the resizing decision is taken early enough such that the rest of the execution is not altered. In this case the reactivity of the hypervisor is high and this may imply a certain overhead that cancels a part of the performance improvement generated by the resizing.

4.4.4 The behavior of the Hypervisor's management policies

In the following section we evaluate different resizing policies and study their behavior when executing the second scenario introduced in Section 4.4.1, composing two Cholesky

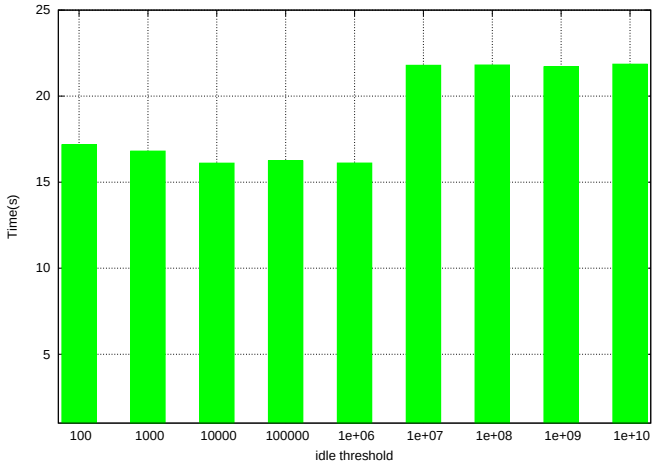
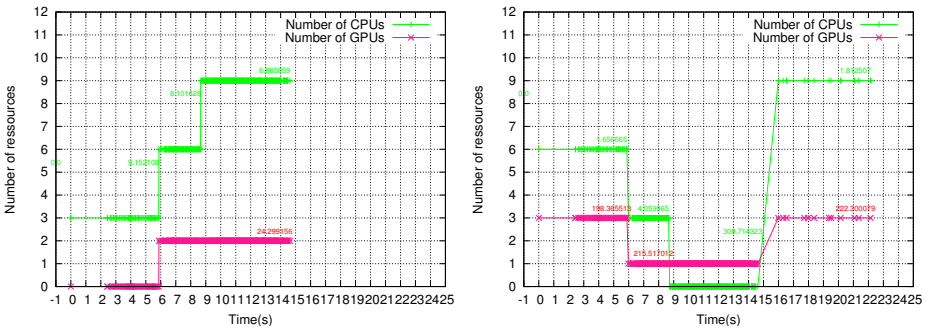


Fig. 6 Choosing the idle threshold to trigger resizing.

factorizations in an arbitrary way. We set the value of the idleness parameter to 100000 (this value was extracted from Figure 6) and compare reactivity and correctness of the resizing decisions for both instant speed based and completion time based strategies.

Instant speed based policy In this experiment, we resize the scheduling contexts whenever the hypervisor decides that there are idle resources. The hypervisor then moves the workers using the instant speed based resizing strategy introduced in Section 3.2.3. We recall that the strategy aims at balancing the instant speed of the contexts.



(a) Resources distribution in Context 1 (b) Resources distribution in Context 2

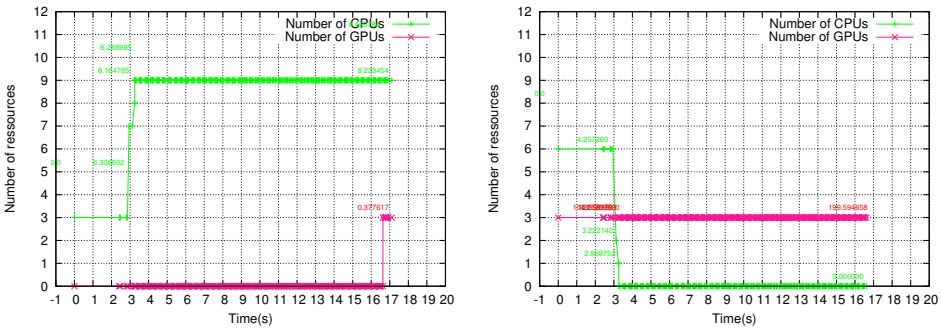
Fig. 7 Resource distribution over the contexts when using the instant speed based policy.

Figure 7 presents the evolution of the number of resources of each type of resources in the two contexts during the execution. We see that the decisions regarding the distribution

of resources do not lead to the optimal distribution of resources, that is 9 CPUs to the first context (small factorization) and 3 GPUs to second one (large factorization). This is mainly due to the fact that trying to balance the speed of the two contexts leads to increase the number of resources allocated to the small factorization which negatively impacts the execution of the large factorization. Furthermore, giving GPUs to the small factorization is a very bad decision since it cannot exploit the GPU at its full potential due to the small granularity of tasks. In this case, the hypervisor performs many resizings which progressively converge toward a good distribution resources.

Once again, having some information regarding the workload of each context would lead to better decisions. For instance, knowing the fraction of the total workload corresponding to each context would give a long term view to the strategy and make it more efficient.

Completion time based policy In this section, we consider the same application as the one used in the previous section. Furthermore, we suppose that the application is able to provide an approximation of the workload of the contexts which allows us to minimize the global makespan time of the application using the completion time based strategies introduced in Section 3.2.3.



(a) Resources distribution in Context 1 (b) Resources distribution in Context 2

Fig. 8 Resource distribution over the contexts when using the completion time based policy.

Figure 8 presents the evolution of the number of resources of each type in the two contexts during the execution. We observe that having information regarding the future of the execution of the application improves the decisions of the hypervisor. By analyzing the speed of the resources during the execution, the correct distribution is achieved gradually. We see in the graphic that executing the algorithm twice is enough to find the best distribution.

Our main objective is to determine the correct distribution of resources in a minimum amount of time. In all the previous situations, the distribution progressively converges to the best solution. However, more complicated applications may require more time to converge to the best distribution. In this case, having precise inputs (e.g. type of tasks) from the application may help the hypervisor execute fewer computations and reach the best solution faster.

4.4.5 Improving efficiency with the programmer's intuition

Further on, we show that the hypervisor can use the programmer's input in order to better react to the irregular parallelism of applications. We show that when the programmer can provide rich information about the application behavior, the hypervisor can improve the overall performance.

We present a scenario where the same application used for the previous experiments is involved. However, in this case we start two different streams of parallel kernels. The first one is composed of three consecutive Cholesky factorizations on matrices of 30 000 x 30 000 elements (using a block size of 960 x 960 elements) and the second one is composed of three factorizations on matrices of 15 000 x 15 000 elements (using a block size of 192 x 192 elements). In this scenario, the first stream is executing efficiently over the entire set of resources and from time to time the second stream steps in and interferes with the parallelism of the first one.

We compare two different situations. In the first one the small stream is submitted to the same context, that contains all the resources, as the first one. In the second situation, initially all the platform is used by the large stream context and the small stream context is activated at some points of the execution of the large stream. When the small stream starts a new parallel kernel, the application tells the hypervisor that the corresponding context needs (resp. releases) some resources (4 CPUs) at the beginning (resp. end) of the kernel which leads to a resources redistribution. It is important to emphasize the fact that at the beginning no specific resources where assigned to the small stream context.

	Execution time
Overlapping contexts	19.7 s
Application driven resizing	17.2 s

Table 5 Application driven resizing policies

In Table 5 we notice an improvement of 2 seconds by resizing dynamically the contexts when the small stream steps in. We see that leaving the two streams blend over the same resources has an important impact on the performance of the overall application. Thus, by assigning periodically some resources to the small stream implies that the cache management of the large stream is affected only when the small one starts a parallel kernel.

5 Related Work

This section discusses related work on runtime systems for shared memory and hybrid CPU+GPU machines, with a particular emphasis on the potential for parallel code composability of the considered environments.

5.1 Runtime systems for multicore machines

OpenMP [4] is probably the most portable way to write programs for shared-memory multiprocessor machines, notably because the number of threads involved in parallel regions can be determined by the underlying runtime system. Unfortunately, most implementations are

not able to adapt the number of threads per region according to the number of coexisting parallel regions (eg. nested parallelism). Thus, they are not able to avoid the oversubscription problem when dealing with several parallel regions simultaneously.

Task-based programming models, such as Cilk [14] or Intel TBB [11], have a greater potential for composability. Intel has been tackling this composition problem on multicore machines for several years, mainly by building their different environments (Intel TBB, Intel Cilk Plus) on a common runtime system basis [24,21]. Sharing the underlying task scheduler allows their environments to run concurrently without causing thread oversubscription. Since TBB 3.0 [21], master threads (i.e. threads running the application code) are isolated from each other thanks to *arenas*. The pool of workers is dynamically split between arenas, proportionally to their requested workers. In the presence of tasks with different priorities (low, normal, high), the scheduler first assigns workers to the highest priority arenas. It is interesting to observe that Intel TBB and Intel OpenMP do not compose well together [24].

Lithe [22] is a runtime system that enables interoperability between different parallel runtimes, e.g. Intel TBB and OpenMP. Lithe is a resource sharing management interface that defines how *harts* (i.e. abstraction of hardware threads) are transferred between parallel libraries within an application. Lithe imposes a hierarchical organization between libraries as well as a specific implementation of multitasking.

5.2 Adaptive scheduling with parallel feedback

Several adaptive scheduling techniques have been studied to efficiently allocate cores to co-running malleable jobs [1,26,25]. Like our approach, these techniques rely on a two-level framework where an Operating System allocator periodically assigns cores to each job using feedback from the hardware, such as the number of wasted processor cycles that occurred in the last period. Our work differs from these techniques because we consider heterogeneous architectures and because we use high-level informations (amount of flops, type of tasks) that allow to benefit from linear programming techniques.

5.3 Runtime systems for accelerator-based platforms

A number of compilation frameworks from various vendors and open source communities have been developed to automatically generate GPU code out of annotated sequential source code. These frameworks rely on runtime systems that provide either very basic offloading services or more sophisticated task scheduling and memory caching services.

Among these runtime systems, we can mention the extension of Charm++ which can handle GPUs [19], Harmony [12] which schedules translated CUDA code on various devices (including CPUs), Qilin [20] which provides a interface to submit kernels that operate on arrays which are automatically dispatched between processing units. Some other runtime systems are based on task data flows parallelism. DAGuE/Parsec [8] and KAAPI [18] are based on a work stealing scheduler, whereas the Anthill extension for GPUs [28,17,27] is based on demand driven scheduler. In OmpSs [7] and StarPU [6], schedulers are considered as plugins. However, these runtimes are based on online scheduling strategies that take affinity into account and have various optimizations based on auto-tuning, data prefetching or work partitioning techniques. Although the aforementioned runtime systems are not subject to resource oversubscription and take task affinities into account, they do not provide

isolation between kernels and they do not support multiple co-existing schedulers within an application.

OpenCL [16] is a standard that provides a unified and portable programming interface for multi-core and accelerator-based architectures. OpenCL provides programmers with a tight control over the utilization of processing units by means of *contexts*. Moreover, the *device fission* feature of OpenCL 1.2 can allow a set of sub-devices to be created, each with its own command queue. This allows applications to dispatch kernels to the various sub-devices as needed. However, devices belonging to different platforms (i.e. device vendors) cannot be placed in the same context, and thus cannot share data buffers. Moreover, there is no notion of scheduling kernels on top of pool of devices in OpenCL.

6 Conclusion and Future Work

To enable high performance computing applications to exploit multiple parallel libraries simultaneously, we introduce *Scheduling Contexts* that allow programmers to control how resources are used by parallel libraries. Contexts can dynamically expand or shrink, and the resource redistribution is triggered by a configurable hypervisor that monitors what happens inside each context. We validate the relevance of our approach by conducting several experiments that emphasize how the dynamic resizing of contexts can lead to a better usage of computing resources. We think that this work brings new insights about how the degree of parallelism of kernels can be auto-tuned to better exploit modern multi-core machines.

In the future, we plan to further investigate new metrics to better guide the redistribution of heterogeneous resources between contexts, including hints provided by developers of parallel libraries.

We also plan to extend our platform for embedded systems (such as heterogeneous multi-core devices used in some handheld devices) where some applications feature parallel kernels with strongly different execution requirements: some computations have to obey real-time constraints while others are requested to achieve a low level of power consumption. The problem of dynamic allocation of computing resources in such systems requires new investigations regarding the algorithms used by the hypervisor.

Finally, we plan to generalize our work to several other task-based runtime systems. As in Lithe [22], our system would be able to concurrently schedule StarPU, OpenMP or Intel TBB-powered parallel libraries.

Acknowledgments

This work was supported by the European Commission as part of the FP7 Project PEPPHER under grant 248481, by the ANR through the COSINUS (PROHMPT ANR-08-COSI-013 project) and CONTINT (MEDIAGPU ANR-09-CORD-025) programs. We thank NVIDIA and its Professor Partnership Program for their hardware donations. We are grateful to Olivier Beaumont for his help regarding the linear programs of the paper.

References

1. Agrawal, K., He, Y., Hsu, W.J., Leiserson, C.E.: Adaptive scheduling with parallelism feedback. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, pp. 100–109. ACM, New York, NY, USA (2006). DOI 10.1145/1122971.1122988. URL http://doi.acm.org/10.1145/1122971.1122988

2. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., Tomov, S.: A hybridization methodology for high-performance linear algebra software for GPUs. in GPU Computing Gems, Jade Edition **2**, 473–484
3. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects
4. ARB, T.O.: The openmp® api specification for parallel programming (2012). Http://openmp.org/
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Proceedings of the 15th Euro-Par Conference, pp. 863–874. Delft, The Netherlands (2009)
6. 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**, 187–198 (2011). DOI 10.1002/cpe.1631. URL http://hal.inria.fr/inria-00550877
7. Ayguadé, E., Badia, R., Igual, F., Labarta, J., Mayo, R., Quintana-Ortí, E.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, pp. 851–862. Springer-Verlag (2009)
8. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: Dague: A generic distributed dag engine for high performance computing. Parallel Computing **38**(Issues 1–2), 37 – 51 (2012). DOI 10.1016/j.parco.2011.10.003. URL http://www.sciencedirect.com/science/article/pii/S0167819111001347
9. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IISWC, pp. 44–54. IEEE (2009)
10. Corporation, I.: MKL reference manual. http://software.intel.com/en-us/articles/intel-mkl. URL http://software.intel.com/en-us/articles/intel-mkl
11. Corporation, I.: TBB reference manual. http://threadingbuildingblocks.org. URL http://threadingbuildingblocks.org
12. Diamos, G.F., Yalamanchili, S.: Harmony: an execution model and runtime for heterogeneous many core systems. In: HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing, pp. 197–200. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1383422.1383447
13. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2), 216–231 (2005)
14. Frigo, M., Leiserson, C., Randall, K.: The implementation of the cilk-5 multithreaded language. SIGPLAN Not. **33**(5), 212–223 (1998). DOI http://doi.acm.org/10.1145/277652.277725
15. Genovese, L., Videau, B., Ospici, M., Deutsch, T., Goedecker, S., Méhaut, J.F.: Daubechies wavelets for high performance electronic structure calculations: The bigdft project. Compte Rendus Mecanique **339**(Issues 2-3), 149 – 164 (2011). DOI http://dx.doi.org/10.1016/j.crme.2010.12.003. URL http://www.sciencedirect.com/science/article/pii/S1631072110002135
16. Group, T.K.: OpenCL - the open standard for parallel programming of heterogeneous systems (2011). Http://khronos.org/opencl/
17. Hartley, T.D.R., Saule, E., Çatalyürek, Ü.V.: Improving performance of adaptive component-based dataflow middleware. Parallel Computing **38**(6-7), 289–309 (2012)
18. Hermann, E., Raffin, B., Faure, F., Gautier, T., Allard, J.: Multi-gpu and multi-cpu parallelization for interactive physics simulations. In: P. D'Ambra, M. Guarracino, D. Talia (eds.) Euro-Par 2010 - Parallel Processing, *Lecture Notes in Computer Science*, vol. 6272, pp. 235–246. Springer Berlin / Heidelberg (2010)
19. Jetley, P., Wesolowski, L., Gioachin, F., Kalé, L.V., Quinn, T.R.: Scaling hierarchical n-body simulations on gpu clusters. In: SC, pp. 1–11. IEEE (2010)
20. Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, pp. 45 –55 (2009)
21. Marochko, A.: Tbb 3.0 task scheduler improves composability of tbb based solutions. (2012). Http://software.intel.com/en-us/blogs/2010/05/13/tbb-30-task-scheduler-improves-composability-of-tbb-based-solutions-part-1/
22. Pan, H., Hindman, B., Asanović, K.: Composing parallel software efficiently with lithe. SIGPLAN Not. **45**, 376–387 (2010). DOI http://doi.acm.org/10.1145/1809028.1806639. URL http://doi.acm.org/10.1145/1809028.1806639
23. Quintana-Orti, G., Igual, F.D., Quintana-Orti, E.S., van de Geijn, R.: Solving dense linear algebra problems on platforms with multiple hardware accelerators. FLAME Working Notes **flawn32** (2008)
24. Sabahi, M.: Getting code ready for parallel execution with intel® parallel composer (2012). Http://software.intel.com/en-us/articles/getting-code-ready-for-parallel-execution-with-intel-parallel-composer

25. Sun, H., Cao, Y., Hsu, W.J.: Efficient adaptive scheduling of multiprocessors with stable parallelism feedback. Parallel and Distributed Systems, IEEE Transactions on **22**(4), 594–607 (2011). DOI 10.1109/TPDS.2010.121
26. Sun, H., Hsu, W.J.: Adaptive b-greedy (abg): A simple yet efficient scheduling algorithm. In: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on Parallel and Distributed Systems, pp. 1–8. Miami, Florida USA (2008). DOI 10.1109/IPDPS.2008.4536546
27. Teodoro, G., Hartley, T.D.R., Çatalyürek, Ü.V., Ferreira, R.: Optimizing dataflow applications on heterogeneous environments. Cluster Computing **15**(2), 125–144 (2012)
28. Teodoro, G., Sachetto, R., Sertel, O., Gurcan, M., Meira, W., Catalyurek, U., Ferreira, R.: Coordinating the use of gpu and cpu for improving performance of compute intensive applications. In: Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on Cluster Computing, pp. 1–10. New Orleans, Louisiana, USA (2009). DOI 10.1109/CLUSTR.2009.5289193
29. Tian, K., Jiang, Y., Shen, X., Mao, W.: Optimal co-scheduling to minimize makespan on chip multiprocessors. Lecture Notes Computer Science **7698** (2013)
30. Tomov, S., Nath, R., Ltaief, H., Dongarra, J.: Dense linear algebra solvers for multicore with gpu accelerators. In: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on Parallel and Distributed Systems, pp. 1 –8. Atlanta, Georgia, USA (2010). DOI 10.1109/IPDPSW.2010.5470941
31. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. Parallel and Distributed Systems, IEEE Transactions on **13**(3), 260–274 (2002). DOI 10.1109/71.993206
32. Van Zee, F., Chan, E., van de Geijn, R., Quintana, E., Quintana-Orti, G.: Introducing: The Libflame library for dense matrix computations. Computing in Science Engineering **PP**(99), 1 (2009). DOI 10.1109/MCSE.2009.154