Composing multiple StarPU applications over heterogeneous machines: a supervised approach

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

Composing multiple StarPU applications over heterogeneous machines: a supervised approach

A.-E Hugo A. Guermouche P.-A. Wacrenier R. Namyst
INRIA, LaBRI, University of Bordeaux
Talence, France

***Abstract*—Enabling HPC applications to perform efficiently when invoking multiple parallel libraries simultaneously is a great challenge. Even if a single 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%).

I. 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 [12], OpenMP or Intel TBB [9] for multicore computers, Anthill [23], DAGuE [6], Charm++ [17], Harmony [10], KAAPI [16], Qilin [18], StarPU [4] or StarSs [5] for heterogeneous configurations) has allowed programmers to rely on thread/task facilities to develop efficient implementations of parallel libraries (e.g. Intel MKL [8], FFTW [11]). The MAGMA library [25], that provides Linear Algebra algorithms on heterogeneous hardware by relying on the StarPU runtime system to perform dynamic scheduling between CPUs and GPUs, well illustrates this trend toward delegating 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 [13]. 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 [20], [19]. 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 have a poor scalability on a high number of nodes. 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 [9].

Moreover, some advanced environments allow for partitioning hardware resources, following an approach similar to virtual machines [20]. 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 multiple 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 [4]. 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

II. Background and Related Work

This sections provides background about 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.

A. Runtime systems for multicore machines

OpenMP [3] 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 [12] or Intel TBB [9], 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 [22], [19]. Sharing the underlying task scheduler allows their environments to run concurrently without causing thread oversubscription. Since TBB 3.0 [19], 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 [22].

Lithe [20] 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.

B. 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 [17], Harmony [10] which schedules translated CUDA code on various devices (including CPUs), Qilin [18] 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 dataflow parallelism. DAGuE [6] and KAAPI [16] are based on a work stealing scheduler, whereas the Anthill extension for GPUs [23], [15], [24] is based on demand driven scheduler. In OmpSs [5] and StarPU [4], 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 aforementionned 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 [14] 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) can not 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.

III. StarPU and the composition problem

A. The StarPU Runtime System

The StarPU runtime system [4] 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.

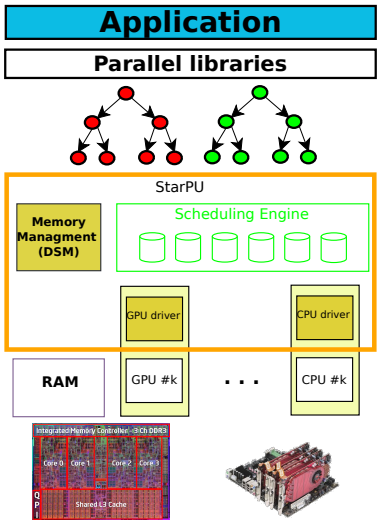

Fig. 1. The architecture of the StarPU runtime system.

B. 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 [26]. 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.

C. 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 issue to reach 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 false dependencies between tasks. 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 can not be controlled anymore by the programmer.

D. 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. However, the complexity of this task is so high that, in the general case, there is no known automatic method that would be able to generate such a super-scheduler out of a given set of arbitrary schedulers. Moreover, scheduling policies may be so diverse that the optimization criteria would be different (e.g. time to completion, power consumption, etc.) In such situations, 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.

IV. Our Approach to Co-scheduling Multiple Parallel Codes

In this section, we describe our generic approach to tackle the issues described in the previous sections. Instead of trying to design a super-scheduler as discussed previously, we propose a solution were 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 our 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 in "the most appropriate way".

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.

A. Extending StarPU with scheduling contexts

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.

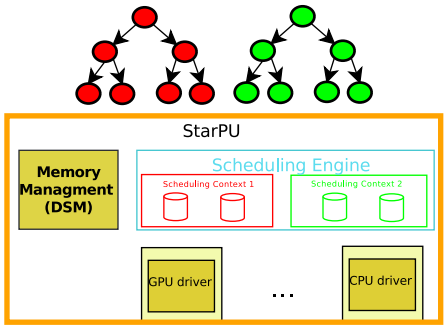

Fig. 2. Scheduling Contexts in StarPU.

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.

The fact that we allow some resources to be shared implies that some schedulers associated with the contexts need to take into account the tasks scheduled on shared resources and coming from other 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.

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 resources 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.

Allocation of processing units

It is worth to note that high performance programmers usually know the characteristics of their kernels and have the ability to analyze and understand the performance of their application.

```
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

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 an algorithm to compute an estimated distribution of resources over the contexts depending on the amount of work (that is, the number of floating point operations) associated with each context. It 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.

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

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.

B. Extending StarPU with the hypervisor

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.

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 management of the contexts and processing units.

The hypervisor (see Figure 4) can be either invoked directly by the application (by creating, destructing or modifying a context during the execution of the application) or triggered periodically according to the behavior of the application.

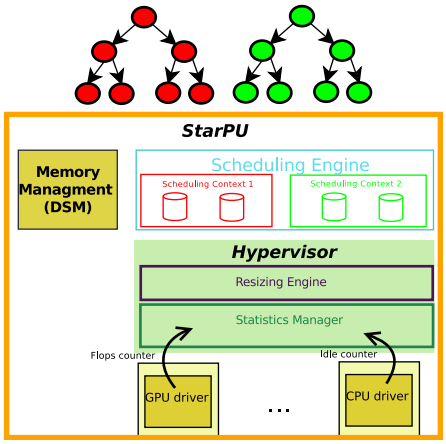

Fig. 4. Placing the Hypervisor in StarPU

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 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 taking decisions about redistribution of resources based on performance forecasts of the execution of the application.

A small number of non-intrusive callbacks are introduced in order to trigger the resizing of certain contexts when the characteristics of the application no longer match the requirements.

Hypervisor policies

The hypervisor can use two different metrics when resizing scheduling contexts. The first one is based on a low-level counter (resource idleness) whereas the second one is the instant speed of the contexts, computed using a mixture of hints coming from the application. In the *Idleness-based resizing* policy, contexts are resized whenever one of their resources is considered to be idle for a period longer than the one specified by the programmer.

In the *Speed-based resizing* policy, the application provides the so called hints which consist in this case in an estimation of the total amount of work corresponding to each parallel kernel and to each of their tasks. The amount of work may represent the number of Flops computed by the programmer or just a coefficient indicating the workload of a task relative to the one of the application. Using this information, the hypervisor computes the instant speed of each parallel kernel and estimates a completion time for each kernel. When the difference of instant speed between the contexts is too high, the hypervisor resizes the contexts in order to minimize the makespan of the whole application. More precisely, when a redistribution is needed, we solve the linear program described by the Equation (1) in order to update the set of resources associated with each context. We believe that the more the runtime system has inputs from the application, the better it will behave. In case no such input is available, either because the application is very irregular or because the programmer is not an expert, strategies such as the *Idleness-based* one can be used, as they are only based on hardware counters. However, it is important to note that even if the workload is unpredictable for the whole execution like for n-body applications, it is often possible to get the workload for the current phase of the computations (current time step, current iteration, ...).

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.

```
/* 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.

V. Evaluation

In this section, we present experiments that evaluate the impact of using scheduling contexts with applications which require multiple parallel kernels to be executed concurrently. For example, 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.

A. 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.

B. Experimental setup

We focus on parallel applications running simultaneously several StarPU-enabled parallel libraries. We perform measurements on parallel kernels from the MAGMA library [25], [2]. 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 [1] 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 used 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 [7]. 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 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 coming from the use of our scheduling contexts, using MAGMA, CFD, or both of them. We show that scheduling contexts are a tool to better respond to the specificity of the kernels. Certain kernels require more resources than others because they can generate more parallelism and exploit more devices more efficiently. Some kernels are not able to efficiently exploit certain types of devices. Therefore, they should execute on the resources where they perform the best.

Even though the experimental applications are synthetic ones, we obtain some preliminary results that confirm the relevance of our approach. The main library we used for our experiments (c.f. the MAGMA library) is the most efficient (together with FLAME [27], [21]) 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 our contexts by confronting them to artificial but well-controlled scenarios that mimic the configurations that can be met within these complex applications. In our experiments, the different parallel kernels (which are in our case task graphs) compete for the resources.

C. 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 thus that by isolating them we 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 I, an experiment where the CFD kernel has to be executed together with the MAGMA kernel described above.

The scheduling strategy used in this case is aware of the specific needs of CFD and thus it avoids any additional data transfers by restricting the execution of CFD tasks on 2 GPUs.

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

TABLE I
CONCURRENT EXECUTION OF CFD BENCHMARK AND CHOLESKY FACTORIZATION OF THE MAGMA LIBRARY

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 I 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.

D. Scheduling contexts to enforce locality

In this section we show that scheduling contexts help to better exploit data reuse and locality. In Table II, 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 can 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.

	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 II
DATA TRANSFER STATISTICS OF CONCURRENT EXECUTION OF THREE FACTORIZATIONS ON *mirage* PLATFORM

E. Efficient execution of several concurrent parallel kernels

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 III we compare the execution time of the application, when it mixes the kernels into one context to the 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 III
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.

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.

F. Using the hypervisor to improve the efficiency of composed parallel codes

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 the user has a good knowledge of the parallelism of his application necessary to determine the initial distribution of the resources among the contexts, he may need to rely on the runtime system to polish the initial resource distribution via dynamic strategies. 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.

Adjusting processing units distribution over contexts

In the following experiment 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 a negative scenario determined empirically. We simulate an application arriving at a point in its execution where the distribution of the resources is no longer efficient. To do this we use an application 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 empirically.

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 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").

The hypervisor will use then one of the runtime based policies to detect that the application is not executing efficiently enough and to resize the contexts such that the configuration should be adapted to the kernels' needs in term of computing resources. We recall the ideas behind these policies. For the **speed-based resizing policy**, the application specifies the total amount of work corresponding to each parallel kernel together with the number of operations of each task composing them. Using this information the runtime system (i.e. the hypervisor) computes the instant speed and dynamically adjusts the size of the contexts such that the makespan of the whole application is minimized. Thanks to the information coming from the application, we ensure that, when possible, the two kernels will end their execution at the same time (which is our goal in this simple scenario). Concerning the **idleness-based resizing policy**, we recall that in this strategy the hypervisor will redistribute idle resources among the contexts.

	Execution time
Best empirical detected distribution	18.6 s
Arbitrary distribution	24.8 s
Speed-based resized distribution	23.8 s
Idleness-based resized distribution	24.3 s

(a) Two Cholesky Factorisations

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

(b) Cholesky Factorisation and CFD solver

TABLE IV
USING THE HYPERVISOR TO CORRECT UNADAPTED DISTRIBUTION OF RESOURCES

In Table IV(a) we can see that the speed-based resizing policy corrects the behavior of the application but the initial incorrect distribution still has an impact on performances. The time spent to detect that the distribution is not correct cannot be entirely recovered. Given an estimation of the total amount of work we compute roughly the number of CPUs and GPUs needed by each context (see Equation (1)). The solution corresponding to this approach is fast to compute but not very accurate (especially when the workload corresponding to each context is not regular). We determined the linear program is executed in 0.08 ms on our platform *mirage* with 9 CPUs and 3 GPUs, as well as on larger platforms (with 40 CPUs).

The idleness-based resizing policy is having more trouble to find a steady configuration since the policy does not have a global temporal view of the system, and it is not aware of the future behavior of the resources (if they will be idle or not).

However, the contribution of the hypervisor is not entirely valued in this case, due to the great scalability of the Cholesky factorization kernel, but it is a simple validation of its behavior.

We illustrate a second scenario, we execute 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. As expected, this proves to be a very inefficient configuration, as the factorization would need one GPU while CFD has one on which it does not scale. We can see in Table IV(b) that the intervention of the hypervisor is essential. We use our hypervisor to resize the contexts and allocate the unused GPU to the Cholesky Factorization and we notice an impressive increase of performance.

In order to have such a good result we rely on a good calibration of StarPU as well as good scheduling decisions at crucial points of the execution. Thus, the result is spectacular because having a GPU idle is an important waste of computation resources. This scenario is possible due to the specificity of the parallelization of the CFD kernel.

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 usually the programmer has important information to provide and that the hypervisor can exploit this information to improve the performance of the application.

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 V
APPLICATION DRIVEN RESIZING POLICIES

In Table V we can notice an improvement of 2 seconds by resizing dynamically the contexts when the small stream steps in. We can 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.

VI. 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 [20], 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] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. A hybridization methodology for high-performance linear algebra software for GPUs. *in GPU Computing Gems, Jade Edition*, 2:473–484.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. volume Vol. 180.

[3] The OpenMP ARB. The openmp® api specification for parallel programming, 2012. http://openmp.org/.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[5] E. Ayguadé, R.M. Badia, F.D. Igual, J. Labarta, R. Mayo, and E.S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par*, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(Issues 1–2):37 – 51, 2012.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE, 2009.

[8] Intel Corporation. MKL reference manual. http://software.intel.com/en-us/articles/intel-mkl.

[9] Intel Corporation. TBB reference manual. http://threadingbuildingblocks.org.

[10] Gregory F. Diamos and Sudhakar Yalamanchili. 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*, pages 197–200, New York, NY, USA, 2008. ACM.

[11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[12] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[13] Luigi Genovese, Brice Videau, Matthieu Ospici, Thierry Deutsch, Stefan Goedecker, and Jean-François Méhaut. Daubechies wavelets for high performance electronic structure calculations: The bigdft project. *Compte Rendus Mecanique*, 339(Issues 2-3):149 – 164, 2011.

[14] The Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2011. http://khronos.org/opencl/.

[15] Timothy D. R. Hartley, Erik Saule, and Ümit V. Çatalyürek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 38(6-7):289–309, 2012.

[16] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-gpu and multi-cpu parallelization for interactive physics simulations. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 235–246. Springer Berlin / Heidelberg, 2010.

[17] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *SC*, pages 1–11. IEEE, 2010.

[18] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45 –55, dec. 2009.

[19] Andrey Marochko. 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/.

[20] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. *SIGPLAN Not.*, 45:376–387, June 2010.

[21] Gregorio Quintana-Orti, Francisco D. Igual, Enrique S. Quintana-Orti, and Robert van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. *FLAME Working Notes*, flawn32, 2008.

[22] Mark Sabahi. 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.

[23] G. Teodoro, R. Sachetto, O. Sertel, M.N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. 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*, pages 1 –10, 31 2009-sept. 4 2009.

[24] George Teodoro, Timothy D. R. Hartley, Ümit V. Çatalyürek, and Renato Ferreira. Optimizing dataflow applications on heterogeneous environments. *Cluster Computing*, 15(2):125–144, 2012.

[25] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[26] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002.

[27] F Van Zee, E Chan, R van de Geijn, E Quintana, and G Quintana-Orti. Introducing: The Libflame library for dense matrix computations. *Computing in Science Engineering*, PP(99):1, 2009.