Exploiting two-level parallelism by aggregating computing resources in task-based applications over accelerator-based machines

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

Exploiting two-level parallelism by aggregating computing resources in task-based applications over accelerator-based machines

Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, Pierre-André Wacrenier
INRIA, LaBRI, University of Bordeaux, Talence, France
Email: firstname.lastname@inria.fr

***Abstract*—Computing platforms are now extremely complex providing an increasing number of CPUs and accelerators. This trend makes balancing computations between these heterogeneous resources performance critical. In this paper we tackle the task granularity problem and we propose *aggregating several CPUs* in order to execute larger parallel tasks and thus find a better equilibrium between the workload assigned to the CPUs and the one assigned to the GPUs. To this end, we rely on the notion of *scheduling contexts* in order to isolate the parallel tasks and thus delegate the management of the task parallelism to the inner scheduling strategy. We demonstrate the relevance of our approach through the dense Cholesky factorization kernel implemented on top of the StarPU task-based runtime system. We allow having parallel elementary tasks and using Intel MKL parallel implementation optimized through the use of the OpenMP runtime system. We show how our approach handles the interaction between the StarPU and the OpenMP runtime systems and how it exploits the parallelism of modern accelerator-based machines. We present experimental results showing that our solution outperforms state of the art implementations to reach a peak performance of 4.5 TFlop/s on a platform equipped with 20 CPU cores and 4 GPU devices.**

***Index Terms*—Multicore; accelerator; GPU; heterogeneous computing; task DAG; runtime system; dense linear algebra; Cholesky**

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 [1], OpenMP or Intel TBB [2] for multicore machines, PaRSEC [3], Charm++ [4], Harmony [5], KAAPI [6], Qilin [7], StarPU [8] or StarSs [9] for heterogeneous configurations) has allowed programmers to rely on task-based runtime systems and develop efficient implementations of parallel libraries (e.g. Intel MKL [10], FFTW [11]).

Since the increase of cores is a clear trend to boost theoretical performance of processors, the gap between the processing power of individual CPU cores and accelerators (e.g. GPU) is increasing. The main issue encountered when trying to exploit both CPUs and accelerators lies in the fact that these devices have very different characteristics and requirements. Compared to regular CPUs, a GPU for instance is composed of many lightweight cores and requires massive parallelism to hide memory latencies and thus fully tap into its potential performance. As a result, GPUs typically exhibit better performance when executing kernels featuring numerous threads, which we call *coarse grain kernels* in the remainder of the paper. On the contrary, regular CPU cores typically reach their peak performance with fine grain tasks working on a reduced memory footprint. To illustrate this claim, we provide in Figure 1 a performance profile of the matrix product kernel (DGEMM) on two different devices: an Intel CPU Haswell Xeon E5-2680 v3 with 12 cores, and a GPU Nvidia K40M. We can observe that the sequential MKL implementation of the DGEMM kernel reaches its peak performance for matrix sizes greater than 200 while in the case of the cuBLAS kernel, the GPU reaches its peak performance for sizes above 2000.

Unfortunately, runtime systems often consider GPUs or Xeon Phi accelerators as single devices, and treat individual cores equally. And because many applications are parallelized using homogeneous block or tile decomposition, runtime systems' schedulers have to cope with very different durations when executing tasks over single cores or over accelerators, resulting in situations where only a few tasks are assigned to CPUs because of bad scores computed by the performance prediction-based heuristics. As a consequence, task-based applications running on such heterogeneous platforms typically adopt an intermediate granularity, chosen as a trade-off between coarse-grain and fine-grain tasks. A small granularity would indeed lead to poor performance on the GPU side, whereas large kernel sizes would dramatically increase the cost of bad load balancing decisions.

The basic solution used by dense linear algebra libraries [12], [13], [14], [15] is to compute a unique size which will represent the best trade-off. Some approaches relax this constraint and are able to split coarse-grain tasks at run time to generate fine-grain tasks for CPUs [16] and provide a more flexible and efficient approach. In this paper, we investigate a dual approach: instead of splitting tasks, we aggregate computing resources in a way such that coarse grain kernels can efficiently be executed by CPU pools and by exploiting inner task parallelism. We provide in Figure 1 a performance of the DGEMM kernel when using the multithreaded Intel MKL implementation. We can see that by doing so, the performance of the kernel scales almost perfectly for matrix

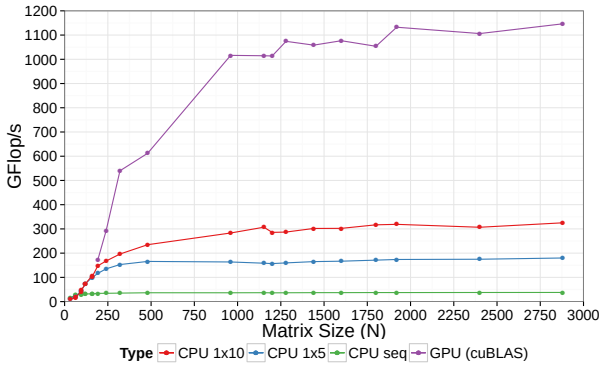


Fig. 1. Performance of the matrix product kernel (DGEMM) on CPU and GPU devices.

sizes larger than 960.

The approach we propose in this paper to tackle the granularity problem is based on resource aggregation: instead of dynamically splitting tasks, we rather aggregate resources to process coarse grain tasks in a parallel fashion. To do so, we introduce a generalization of the concept of scheduling contexts [17] to handle any type of parallel tasks. We illustrate how state of the art scheduling heuristics are upgraded to deal with parallel tasks. We then evaluate our solution on a dense Cholesky factorization kernel and show that it outperforms state of the art implementations to reach a peak performance of 4.5 TFlop/s on a platform equipped with 20 CPU cores and 4 GPU devices.

II. Related Work

A number of research efforts have recently been focusing on redesigning HPC applications to use dynamic runtime systems. As an example, several sparse direct solvers have been redesigned on top of task-based runtime systems [18], [19] and exhibit high performance and improved portability. Similar efforts have also been undertaken in the field of fast multipole method computations [20], [21]. The dense linear algebra community has strongly adopted this modular approach, by relying on task-based runtime systems, over the past few years [12], [13], [14], [15] and delivered production-quality software relying on it. For example, The `MAGMA` library [14], provides Linear Algebra algorithms over heterogeneous hardware and can optionally use the StarPU runtime system to perform dynamic scheduling between CPUs and GPUs, illustrating the 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. However, these approaches require that accelerators process a large share of the total workload to ensure a good load balancing between resources. Additionally, all these approaches require an identical tile size, consequently, all tasks have the same granularity independently from where they are executed leading to a loss of efficiency of both the CPUs and the accelerators.

There are a few prior studies trying to resolve the granularity issue between regular CPUs and accelerators in the specific context of dense linear algebra. Most of these efforts rely on heterogeneous tile sizes [22], [23] which may involve extra memory copies when split data need to be coalesced again [24]. However, in most of these efforts, the decision to split a task is made statically at submission time. More recently, a more dynamic approach has been proposed in [16] where the large granularity tasks are hierarchically split at runtime when they are executed on CPUs. Although this paper successes at tackling the granularity problem, the proposed solution is specific to linear algebra kernels. In the context of this paper, we tackle the granularity problem with the opposite point of view and a more general approach: rather than splitting coarse grained tasks, we aggregate resources which cooperate to process the task in parallel. By doing so, the runtime system does not only exploit sequential tasks but also parallel ones. This requires the runtime system to be able to handle any type of inner parallelism within a given task. For instance, the programmer may provide different implementations of the parallel tasks potentially belonging to different parallel libraries. Nevertheless, these latter may be optimized on top of different runtime systems. Indeed, calling simultaneously several parallel procedures is a difficult matter because usually parallel libraries make the assumption that each procedure has exclusive access to the architecture of the machine. They are not aware of the resource utilization of one another and they may thus oversubscribe threads to the processing units. This problem is referred to as the parallel *composability* problem. This issue has been first tackled within the Lithe framework [25] which is a runtime system enabling interoperability between different parallel runtime systems, e.g. Intel TBB and OpenMP. It is a resource sharing management interface that defines how *harts* (i.e. abstraction of hardware threads) are transferred between parallel libraries within an application. Some efforts have been made within Intel TBB [25], [26], [27] to ensure a good behavior when concurrent parallel TBB kernels are used. These contributions suffered either from the fact that they do not allow to dynamically change the number of resources associated with a parallel kernel or from the fact that they are restricted to a given inner runtime system. Our contribution in this study is a generalization of a previous work [28] where we introduced the so-called scheduling contexts which aim at structuring the parallelism for complex applications relying on concurrent parallel StarPU kernels. We will present in the next section how we extend the scheduling contexts to be able to manage any inner runtime system.

III. Background

Tackling the composability problem at the runtime system level can ensure both the portability and the reusability of the solution. Indeed, such an approach benefits from low level information regarding the architecture of the machine and allows the programmer to have full control on the resource allocation of the application. We integrate our solution inside

the StarPU runtime system as it provides a flexible platform to deal with heterogeneous architectures.

A. The StarPU runtime system

StarPU [8] 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 the runtime system. 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 in different memory locations as long as it is not modified. Asynchronous data prefetching is also used to hide memory latencies.

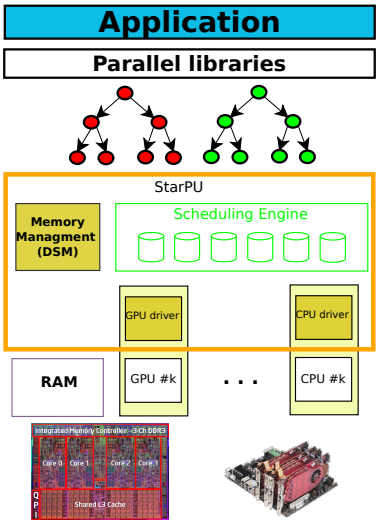


Fig. 2. The architecture of the StarPU runtime system.

StarPU is a platform for developing, tuning and experimenting with various task scheduling policies in a portable way (see Figure 2). It provides an abstract view of the machine by relying on the notion of worker to describe the computing resource (e.g. CPU, GPU) in charge of executing the tasks. Implementing a scheduler consists in creating a set of queues, associating them with the different processing units, and defining the code that is triggered each time a new task becomes 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 [29]. 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.

These models are actually performance history tables dynamically built during the application run. By observing the execution of the tasks, the runtime is able to capture the speedup and the affinities between the tasks and processors. Therefore, even without the programmer's involvement, the runtime can provide a relatively accurate performance estimation of the expected requirements of the tasks allowing the scheduler to take appropriate decisions when assigning tasks to computing resources.

B. Scheduling contexts to compose parallel codes

In order to deal with the composability of parallel codes we introduced in previous work [17] the notion of *scheduling contexts*. They are meant to encapsulate a parallel code and restrict its execution on a section of the machine. Thus, we consider it as an abstract set of resources on which the parallel kernel runs. This allows the programmers to structure the parallelism of their application (since parallel kernels are distributed among the different contexts) in order to dynamically control the distribution of computing resources (i.e. CPUs and GPUs) among scheduling contexts. Therefore scheduling contexts represent a tool for the programmer to control the distribution of the workload of the application but also to provide informations about the parallel structure of the application to the runtime. Thanks to this information the runtime is able to perform a better mapping of the workload on the underlying topology. Thus, different optimizations can be made depending on the parallel structure of the application and the underlying hardware architecture. For instance, scheduling contexts can be used to enforce the locality by allocating only resources sharing a NUMA node to a certain kernel. However, our previous contribution was restricted to parallel kernels which are based on the StarPU runtime system.

IV. A runtime solution to deal with nested parallelism

We now explain how we extend the concept of scheduling contexts to deal with parallel codes implemented on top of other runtime systems. Scheduling contexts can be considered as isolation containers that host a runtime system which will not interfere with the outside. It is a way to conveniently abstract the notion of computing resource by hiding the actual number of physical computing resources (ie. workers) assigned to contexts. Thus, in the remainder of the paper we refer to these containers as *clusters* where the main scheduler is only in charge of dispatching tasks to. If the inner scheduler (or runtime system) of the task is a StarPU one or a SPMD one (i.e. a call done by each worker executing the parallel task) it can use directly the workers assigned to the context and distribute the tasks accordingly. However, if it is not the case, the inner scheduler may have its own set of threads. In this

case, when creating the scheduling context the StarPU workers are paused allowing the parallel code to have its own set of threads and avoiding oversubscribing threads to the resources.

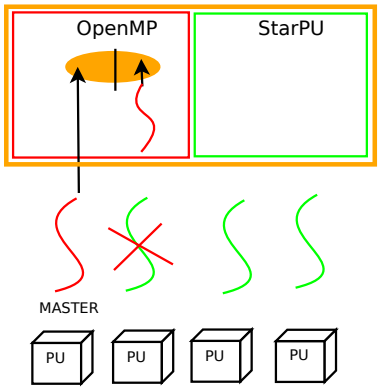


Fig. 3. Scheduling contexts isolating OpenMP parallel tasks from StarPU parallel tasks

In Figure 3 we illustrate a scenario where we compose a parallel kernel relying on the OpenMP runtime system and another code implemented on top of StarPU. Each parallel section is isolated inside a scheduling context. The inner scheduler of the OpenMP runtime system creates its own team of threads. In this case, allowing the execution of both StarPU and OpenMP set of threads would be unnecessary and may prevent us from obtaining good performance.

Thus we allow only one StarPU worker to keep running. This latter called the *master worker* of the scheduling context, is in charge of popping the task assigned to the context by the outer scheduler. Then it takes the role of a regular application thread with respect to the inner runtime system when tasks has to be executed. Coming back to Figure 3, the green threads represent the StarPU workers and the red ones the OpenMP threads. The master worker joins the team of OpenMP threads and the green thread pinned on the same core than the OpenMP slave thread is paused.

Nevertheless, depending on the inner scheduling engine one may be able to remove (resp. add) the computing resources at any time of the execution of a parallel task. However, this requires the inner scheduler (resp. runtime system) to be able to support such an operation. For instance, parallel kernels implemented on top of runtime systems like OpenMP will not allow removing a computing resource during the execution of the parallel task. In this case we refer to the corresponding parallel task as a *moldable* one and we consider resizing the corresponding scheduling context only at the end of such a task or before starting a new one.

A. Scheduling policies for parallel tasks

Parallel tasks may be explicitly assigned to clusters or automatically through different scheduling strategies. For our experimental validation we use the StarPU scheduling platform that we extend in order to transparently assign tasks to both regular workers and scheduling contexts.

One of the StarPU policies which is adapted to the heterogeneous architectures is the MCT scheduler. It is a task-centric scheduler which assigns ready tasks to the computing resources that will end their execution at the earliest. Thus, the policy relies on a history based performance model to estimate the execution time of the task on a given resource. We adapted this strategy and more precisely the underlying performance models to be able to select not only a worker but also a cluster as the resource which will execute the task. This time the performance model is parametrized not only by the size and type of the task together with the candidate resource but also by the size of the resource if the latter is a cluster. Thus, the policy can choose to which worker or context a task is assigned depending on where it would finish first. This approach allows dealing with a heterogeneous architecture made of different types of processing units as well as clusters grouping different sets of processing units.

We consider an example in Figure 4 where we trace a part of the execution of a Cholesky Factorization kernel. On the x-axis we have the time and each line corresponds to a CPU (the first lines) or GPU (the last two lines). Each color corresponds to a context. The global context CTX 0 has tasks whose execution is traced in white. The contexts CTX 1 (green), CTX 2 (yellow) and CTX 3 (blue) aggregate each 6 CPUs and will execute parallel tasks relying on a non-StarPU inner scheduler. The context CTX 0 will thus use an MCT policy in order to schedule tasks on either of the three contexts (CTX 1, CTX 2 and CTX 3) or on one of the two GPUs. Assigning a parallel task to a cluster consists in moving the task from the global context to the corresponding context, in this case the color of the task changes from white to green/yellow/blue in order to show to which context it was actually assigned.

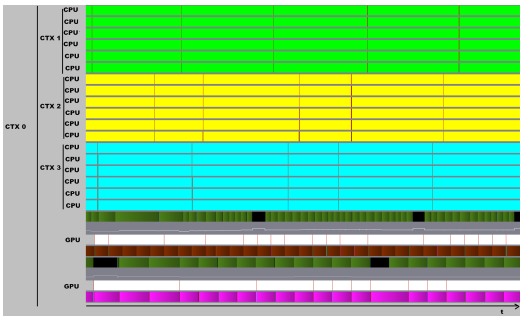


Fig. 4. Section of a Cholesky factorization using the parallel Intel MKL library for the elementary tasks - MCT on 3 contexts and 2 GPUs

```c
int main()
{
  int cpus[3] = {CPU_1, CPU_2, CPU_3};

  /* define the table of resource ids */
  ctx = starpu_create_sched_ctx(cpus,3);

  /* submit the set of tasks to the context*/
  for( i = 0; i < ntasks; i++)
  {
    /* indicate the prologue callback */
    tasks[i].prologue = cl_prologue;

    /* indicate the codelet of the task*/
    tasks[i].codelet = codelet;

    /* submit the task to the context */
    starpu_task_submit_to_ctx(tasks[i], ctx);
  }
}
```

```c
#include "mkl.h"

void cl_prologue()
{
  /* context on which the task executes */
  int ctx = starpu_get_current_ctx();

  /* get the CPUs of the current context */
  int cpus[MAX_WORKERS];
  int ncpus = starpu_get_cpus(ctx, cpus);

 /* bind openmp threads to CPUs */
#pragma omp parallel num_threads(ncpus)
  bind_to_cpu(cpus[omp_get_thread_num()]);
}

void codelet_cpu_func()
{
  /* call the mkl parallel kernel */
  DGEMM(...);
}
```

Fig. 5. Executing Intel MKL parallel codes within a scheduling context

B. Case Study: parallel version of the Intel MKL library

In this context, we illustrate how the proposed framework can deal with parallel tasks by taking a specific example, namely the parallel multithreaded version of the Intel MKL library. First of all, we recall that scheduling contexts allow the schedulers to run unmodified as guests in an isolated manner. In fact they are only informed on which resources they are allowed to execute and then they assign the workload according to their algorithm. However, Intel MKL parallel library requires executing an inner runtime specific code in order to enforce the required resource allocation.

In order to deal with this requirement we rely on a StarPU functionality that allows a code provided by the programmer, which will be refered to as a *callback*, to be executed at different points of the execution of a given task. One such callback is the *prologue callback* which is executed by the master worker when creating the context. We integrate in this callback the specific code required by Intel MKL such that it is transparently triggered before starting executing any sequence of parallel tasks.

We show in Figure 5 an example of how this tool can be used. Our aim is to typically isolate and bind the execution of Intel MKL parallel tasks on specific parts of the machine. Therefore, we provide the implementation of the prologue callback that consists in creating the OpenMP team of threads needed by the kernel and binding them to the logical ids on which the corresponding scheduling context is allowed to execute. Further on, when executing the Intel MKL implementation of the parallel task, the inner scheduler reuses the previously created threads, that are correctly fixed on the required computing resources.

This approach can be generalized to other runtime systems as the programmer can provide the implementation of the prologue callback and thus use the necessary functions in order to indicate the resource allocation required for the corresponding scheduling context. Such a runtime should however respect certain properties: be able to be executed on a restricted set of resources and allow the privatization of its global and static variables.

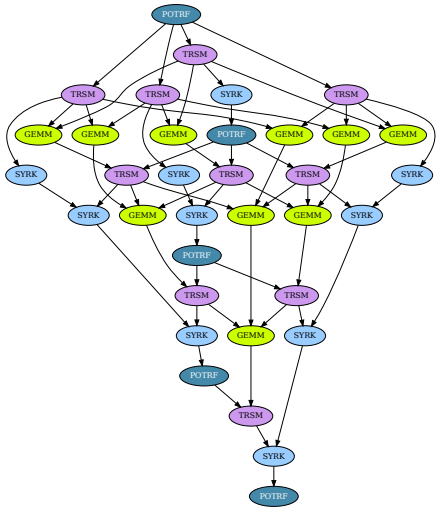


Fig. 6. Task diagram of a Cholesky factorization

V. Experimental Results

We evaluate our method on a Cholesky factorization, an application widely used in linear algebra. This algorithm is present in multiple linear algebra libraries [14], [15], which represent the computations as a DAG of tasks. We present in Figure 6 the DAG of the Cholesky factorization on a matrix containing 5x5 tiles.

For our evaluation, we use the Cholesky factorization of `Chameleon`[1] [30], a dense linear algebra library for heterogeneous platforms based on the StarPU runtime system. Similarly to most task-based linear algebra libraries, `Chameleon` relies on a BLAS library for optimized implementations of the kernels (cuBLAS for NVidia GPUs, MKL for Intel CPUs

[1] https://project.inria.fr/chameleon/

and Co-Processors). Our adaptation of `Chameleon` does not change the high level task-based algorithms and subsequent DAGs. We simply extend the prologue of each task so as to use the OpenMP implementation of MKL inside contexts, as depicted in section IV-B. We call `pt-Chameleon` this adapted version of `Chameleon` that handles parallel tasks.

In order to validate our method, we also compare our `pt-Chameleon` Cholesky factorization to the one of the state of the art `MAGMA` library.

We conducted our experiments on the PlaFRIM experimental testbed[2]. The machine we use is heterogeneous and composed by **two 12-cores Intel Xeon CPU E5-2680 v3** (@2.5 GHz equipped with 30 MB of cache each) and enhanced with **four NVidia K40m GPUs**. Each CPU possess a theoretical peak of 480 GFlop/s in double precision while each GPU boasts a theoretical peak of 1.43 TFlop/s.

All experiments have been carried out using the latest version of the StarPU runtime system. We use a recent version of the baseline `Chameleon` library and its adaptation `pt-Chameleon`. We also compare our solution to `MAGMA` version 1.6. Everything was compiled using the Intel compiler version 2015.3.187. We use the same version of Intel MKL for the CPU BLAS implementation and CUDA 7.0.28 for the cuBLAS GPU kernels. Our `Chameleon` and `pt-Chameleon` experiments are conducted using the MCT scheduler implementation of StarPU adapted to the use of parallel tasks as described in section IV-A.

In figures 7 and 9 we show reference lines (horizontal solid lines in the plots) of the peak performance of the most efficient BLAS kernel: DGEMM. The numbers come from the experiment depicted on Figure 1, where DGEMM reaches 700 GFlop/s on 20 cores; and 1.1 TFlop/s on each GPU. These are clearly upper bounds for a Cholesky factorization, which involves several other less efficient BLAS kernels.

We report in figure 7 the performance behavior of several implementations of the Cholesky factorizations: `MAGMA`, `Chameleon`, and `Chameleon` enhanced with our parallel task framework `pt-Chameleon`. We only report the results with a configuration where 2 parallel workers aggregating 10 cores each are used since empirical experiments showed that it is the configuration providing the best performance (we recall that the CPU sockets are composed of 12 cores, from which two are used to control the GPU devices). We can observe that for the single GPU case, `pt-Chameleon` outperforms both `MAGMA` and `Chameleon`. Moreover, we can observe that when the number of GPUs increases, the gap between `pt-Chameleon` and the other implementation is higher. This can be explained by the fact that when increasing the number of GPUs, the tasks executed by regular cores slow down the whole execution whereas when using `pt-Chameleon`, the parallel tasks are more efficient and less penalizing. This is particularly true for the DPOTRF tasks, which are on the critical path of the whole factorization, and perform better on 10 CPU cores than on a GPU (when considering tile size of size less than 2500). This leads to a decrease of the length of the critical path in the case of the `pt-Chameleon` implementation which may lead to performance improvements. Another interesting observation concerns the total workload assigned to CPU cores by the dynamic scheduler : It is almost the same for `Chameleon` and `pt-Chameleon` (a maximal difference of 5% can be observed). Finally, we can see that `pt-Chameleon` is able to deal with coarser granularities leading to performance improvements: the peak performance obtained with 4 GPUs and 20 CPUs is around 4.25 TFlop/s. This illustrates the interest of relying on parallel tasks within a modern runtime system like StarPU to tackle the granularity problem related to the heterogeneity of the resources.

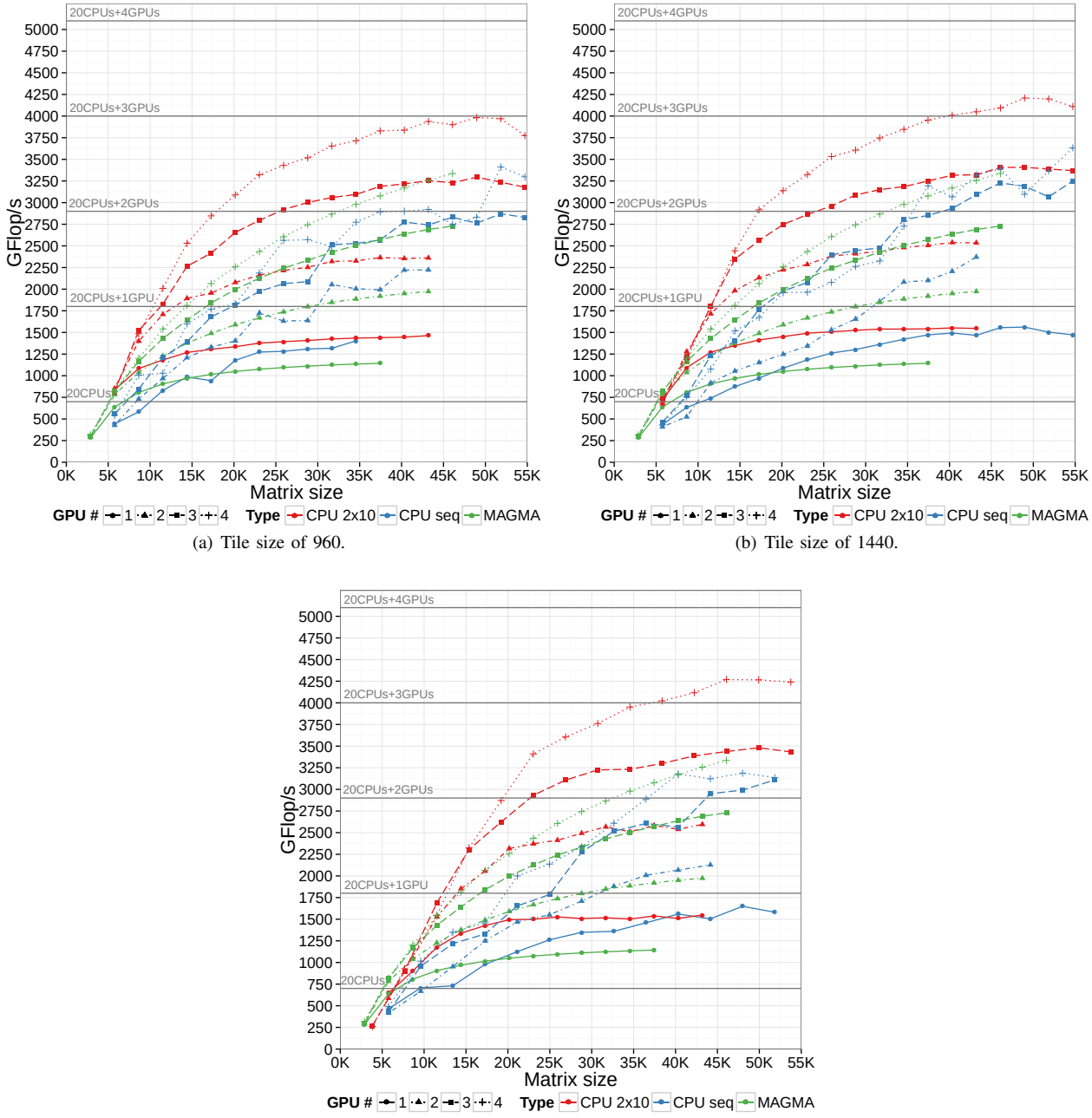
[2]https://plafrim.bordeaux.inria.fr/

	Tile size		
	960	1440	1920
2x10	91%	94%	90%
4x5	90%	76%	54%

TABLE I
LEVEL 3 CACHE HIT RATIOS FOR A CHOLESKY FACTORIZATION OF A MATRIX OF ORDER 46000 RUNNING ON 20 CPUS AND 4 GPUS.

We report in Figure 8 a comparison of the behavior of two different configurations for `pt-Chameleon` clusters: 2 clusters of 10 workers each, and 4 clusters containing 5 workers each (we recall that our test platform is composed by 2 CPUs having 12 cores each). We can see that for the finer granularity (i.e. tile size of 960) both executions have the same behavior and provide the same performance. However, with the increase of the task granularity, either for tile size of 1440 or 1920, the performance obtained with the 4x5 configuration is poor in comparison with the one obtained with the 2x10 configuration. This is mainly due to the fact that in the 2x10 configuration there is only one parallel kernel running on a given processor while in the 4x5 configuration two kernels are competing for the cache which induces a drop in performance when the granularity of the kernels is large. As an illustration, let us consider the Cholesky factorization of a matrix of order 46000 running on 20 CPUs and 4 GPUs. First of all, it is important to emphasize that for all executions reported in Figure 8 the total number of tasks executed by the CPU cores for a given matrix size is the same for both configurations. We provide in Table I the level 3 cache hit ratios for the 2x10 and 4x5 configurations on this factorization. We can see that for moderate granularities, the cache hit ratios are similar for both configurations (leading to equivalent performance). However, with the increase of the tasks granularity the hit ratio of the 4x5 starts to drop to reach 54% for the largest granularity considered while the 2x10 configuration keeps an almost constant level 3 cache hit ratio. This illustrates the fact that the locality is enhanced when relying on the 2x10 configuration which leads to a more stable behavior (the error bars showing observed values across multiple runs provided in Figure 8 illustrate this claim).

Finally, in Figure 9 we consider the execution of the Cholesky factorization on 2 clusters of 10 cores each with a tile



(a) Tile size of 960.

(b) Tile size of 1440.

(c) Tile size of 1920.

Fig. 7. Performance of the Cholesky factorization. The solid horizontal lines represent the peak performance of the DGEMM kernel.

size of 1440. We compare the performance of the application when using two different versions of the MCT scheduler: 1) the version described in Section IV-A, 2) a constrained version where some specific tasks, namely DPOTRF and DTRSM, are executed only on CPU workers. This second strategy is equivalent to what is done in [16] where only DGEMM kernels are executed on GPU devices. We can observe that for small matrices, using the regular MCT scheduler leads to a better performance since in the constrained version the amount of work done by CPUs is too large. However, when the matrix size gets larger, the constrained version starts to be efficient and leads to a 5% increase in performance on average which leads to a peak performance of 4.5 TFlop/s on our test platform.

VI. Conclusion

One of the biggest challenge raised by the design of high performance task-based applications on top of heterogeneous accelerator-based machines lies in choosing the adequate granularity of tasks.

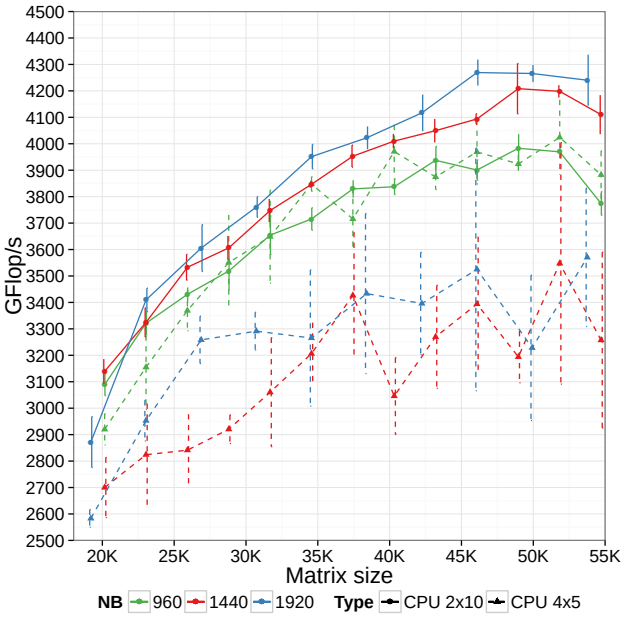


Fig. 8. Comparison of the `pt-Chameleon` Cholesky factorization with two different configurations for the parallel kernels. 20 CPUs and 4 GPUs are used.

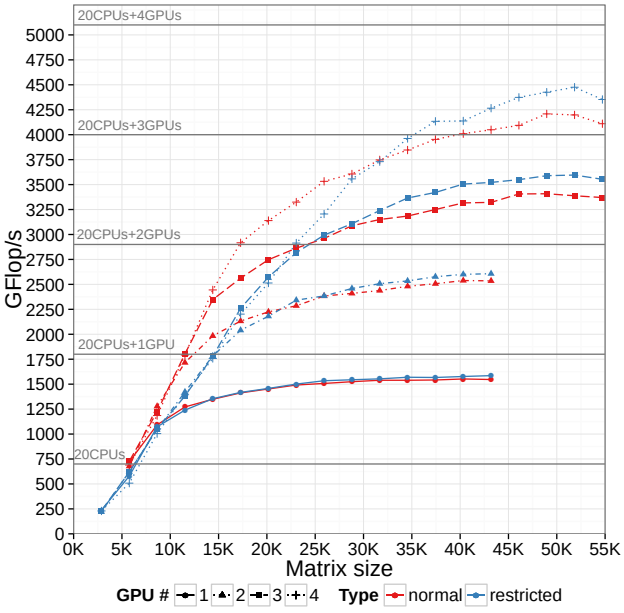


Fig. 9. Comparison of the regular `pt-Chameleon` Cholesky factorization and the constrained `pt-Chameleon`. 20 CPUs and 4 GPUs are used. The tile size is equal to 1440.

Our approach consists in reducing the performance gap between accelerators and single cores by scheduling parallel tasks over clusters of CPUs. The execution of tasks inside such clusters can virtually rely on any thread-based runtime system, and does not interfere with the outer scheduler.

We have extended the concept of scheduling context, which we introduced in a previous work, so that the main runtime system only sees virtual computing resources on which it can schedule parallel tasks (e.g. BLAS kernels).

We present an evaluation of our approach on a regular dense linear algebra kernel. We show that our approach is able to outperform the `MAGMA` state-of-the-art dense linear algebra library on large matrices by using the same task granularity on accelerators and clusters. As a side effect, the overhead of the main scheduler has been strongly improved because of the decrease of the number of computing resources.

In the near future, we intend to further extend this work along several directions. First, we plan to investigate how our approach could be applied to more complex irregular applications (eg. sparse direct solvers or Fast Multipole Methods). On the hardware side, we also have conducted preliminary experiments that suggest our approach can also benefit to manycore accelerators such as the Intel Xeon Phi processor. Indeed, forming small clusters of cores on such a processor decreases cache pressure and helps placing kernels in situations where their performance is actually maximal.

In the long term, we plan to investigate dynamic task granularity adaptation, which is complementary to our approach. We believe mixing both approaches will allow the algorithms and applications designers to have more flexibility regarding the ability to use multiple granularities simultaneously.

Acknowledgment

This work is supported by the Agence Nationale de la Recherche, under grant ANR-13-MONU-0007. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PLaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS.

References

[1] M. Frigo, C. Leiserson, and K. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, 1998.

[2] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *Parallel Computing*, vol. 38, no. Issues 1, pp. 37 – 51, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001347

[4] D. M. Kunzman and L. V. Kalé, "Programming heterogeneous clusters with accelerators using object-based programming," *Scientific Programming*, vol. 19, no. 1, pp. 47–62, 2011.

[5] G. F. Diamos and S. 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*. New York, NY, USA: ACM, 2008, pp. 197–200.

[6] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-gpu and multi-cpu parallelization for interactive physics simulations," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. DAmbra, M. Guarracino, and D. Talia, Eds. Springer Berlin / Heidelberg, 2010, vol. 6272, pp. 235–246.

[7] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, dec. 2009, pp. 45 –55.

[8] 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*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[9] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Euro-Par*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862. [Online]. Available: http://portal.acm.org/citation.cfm?id=1616772.1616863

[10] I. Corporation, "MKL reference manual," http://software.intel.com/en-us/articles/intel-mkl. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl

[11] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[12] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Par. Comp.*, vol. 35, no. 1, pp. 38–53, 2009.

[13] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.*, vol. 36, no. 3, 2009.

[14] 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," vol. Vol. 180.

[15] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra, "Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach," *Scalable Computing and Communications: Theory and Practice*, 2013.

[16] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical dag scheduling for hybrid distributed systems," in *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Hyderabad, India, May 2015.

[17] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst, "Composing multiple starpu applications over heterogeneous machines: A supervised approach," in *IPDPS Workshops*, 2013, pp. 1050–1059.

[18] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Multifrontal qr factorization for multicore architectures over runtime systems," in *Euro-Par 2013 Parallel Processing - 19th International Conference*, 2013, pp. 521–532.

[19] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca, "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes," INRIA, Rapport de recherche RR-8446, Jan. 2014. [Online]. Available: http://hal.inria.fr/hal-00925017

[20] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for multicore architectures," *SIAM J. Scientific Computing*, vol. 36, no. 1, 2014. [Online]. Available: http://dx.doi.org/10.1137/130915662

[21] H. Ltaief and R. Yokota, "Data-driven execution of fast multipole methods," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 11, pp. 1935–1946, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3132

[22] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304625

[23] J. V. F. Lima, F. Broquedis, T. Gautier, and B. Raffin, "Preliminary experiments with xkaapi on intel xeon phi coprocessor," in *25th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2013, Porto de Galinhas, Pernambuco, Brazil, October 23-26, 2013*, 2013, pp. 105–112. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2013.28

[24] K. Kim, V. Eijkhout, and R. A. van de Geijn, "Dense matrix computation on a heterogenous architecture: A block synchronous approach," Texas Advanced Computing Center, The University of Texas at Austin, Tech. Rep. TR-12-04, 2012.

[25] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with lithe," *SIGPLAN Not.*, vol. 45, pp. 376–387, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806639

[26] A. 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/.

[27] Z. Majo and T. R. Gross, "A library for portable and composable data locality optimizations for numa systems," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688509

[28] A. Hugo, A. Guermouche, P. Wacrenier, and R. Namyst, "Composing multiple starpu applications over heterogeneous machines: A supervised approach," *IJHPCA*, vol. 28, no. 3, pp. 285–300, 2014. [Online]. Available: http://dx.doi.org/10.1177/1094342014527575

[29] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, Mar 2002.

[30] 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," *GPU Computing Gems, Jade Edition*, vol. 2, pp. 473–484, 2011.