Resource aggregation for task-based Cholesky Factorization on top of modern architectures

T. Cojean[a], A. Guermouche[a], A. Hugo[b], R. Namyst[a], P.A. Wacrenier[a]

[a] *INRIA, LaBRI, University of Bordeaux, Talence, France*
[b] *University of Uppsala, Sweden*

Abstract

Hybrid computing platforms are now commonplace, featuring a large number of CPU cores and accelerators. This trend makes balancing computations between these heterogeneous resources performance critical. In this paper we propose *aggregating several CPU cores* in order to execute larger parallel tasks and improve load balancing between CPUs and accelerators. Additionally, we present our approach to exploit internal parallelism within tasks, by combining two runtime system schedulers: a global runtime system to schedule the main task graph and a local one one to cope with internal task parallelism. We demonstrate the relevance of our approach in the context of the dense Cholesky factorization kernel implemented on top of the StarPU task-based runtime system. We present experimental results showing that our solution outperforms state of the art implementations on two architectures: a modern heterogeneous machine and the Intel Xeon Phi Knights Landing.

Keywords: Multicore; accelerator; GPU; heterogeneous computing; Intel Xeon-Phi KNL; task DAG; runtime system; dense linear algebra; Cholesky factorization

1. Introduction

Due to recent evolution of High Performance Computing architectures toward massively parallel heterogeneous multicore machines, many research efforts have recently been devoted to the design of runtime systems able to provide programmers with portable techniques and tools to exploit such complex hardware. The availability of mature implementations of such runtime systems (*e.g.* Cilk [1], OpenMP or Intel TBB [2] for multicore machines, APC [3], Charm++ [4], KAAPI [5], Legion [6], PaRSEC [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]).

However one of the main issues encountered when trying to exploit both CPUs and accelerators is that these devices have very different characteristics

Email addresses: terry.cojean@inria.fr (T. Cojean), abdou.guermouche@labri.fr (A. Guermouche), andra.hugo@it.uu.se (A. Hugo), raymond.namyst@u-bordeaux.fr (R. Namyst), pierre-andre.wacrenier@labri.fr (P.A. Wacrenier)

and requirements. Indeed, GPUs typically exhibit better performance when executing kernels applied to large data sets, which we call *coarse grain kernels* (or tasks) in the remainder of the paper. On the contrary, regular CPU cores reach their peak performance with fine grain kernels working on a reduced memory footprint. To work around this granularity problem, task-based applications running on such heterogeneous platforms typically adopt a medium granularity, chosen as a trade-off between coarse-grain and fine-grain kernels. A small granularity would indeed lead to poor performance on the GPU side, whereas large kernel sizes may lead to an under-utilization of CPU cores because (1) the amount of parallelism (*i.e.* task graph width) decreases when kernel size increases and (2) the efficiency of GPU increases while a large memory footprint may penalize CPU cache hit ratio. This trade-off technique is typically used by dense linear algebra hybrid libraries [12, 13, 14]. The main reason for using a unique task granularity in the application lies in the complexity of the algorithms dealing with heterogeneous task granularities even for very regular applications such as dense linear libraries. However some recent approaches relax this constraint and are able to split coarse-grain tasks at run time to generate fine-grain tasks for CPUs [15].

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 manner on the critical resource, the CPU. To deal with Direct Acyclic Graphs (DAGs) of parallel tasks, we have enhanced the StarPU runtime system (see [8, 16]) to cope with parallel tasks, the implementation of which relies on another parallel runtime system (*e.g.* OpenMP). This approach allows us to delegate the division of the kernel between resources to a specialized library. We illustrate how state of the art scheduling heuristics are upgraded to deal with parallel tasks. Although our scheme is able to clusters of arbitrary sizes, we evaluate our solution with homogeneous configurations of fixed-size clusters. We show that using our solution for a dense Cholesky factorization kernel outperforms state of the art implementations to reach a peak performance of 4.6 Tflop/s on a platform equipped with 24 CPU cores and 4 GPU devices. Moreover, we show that exploiting internal task parallelism not only allows to better handle the granularity problem but also offers more flexibility to exploit modern many-core systems. We highlight the limits of standard task-based approaches based on modern systems like `Intel KNL` device and present results illustrating the interest of our approach which reaches a peak performance 1.5 Tflop/s.

2. Related Work

A number of research efforts have recently been focusing on redesigning HPC applications to use dynamic runtime systems. The dense linear algebra community has massively adopted this modular approach over the past few years [12, 13, 14] and delivered production-quality software relying on it. For example, the `MAGMA` library [13], 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 fair load balancing between resources. Additionally, all these approaches rely on an uniform 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.

Recent attempts have been made 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 [17] which may involve extra memory copies when split data need to be coalesced again [18]. However the decision to split a task is mainly made statically at submission time. More recently, a more dynamic approach has been proposed in [15] where coarse grain 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 computing units which cooperate to process the task in parallel. By doing so, our runtime system does not only support sequential tasks but also parallel ones.

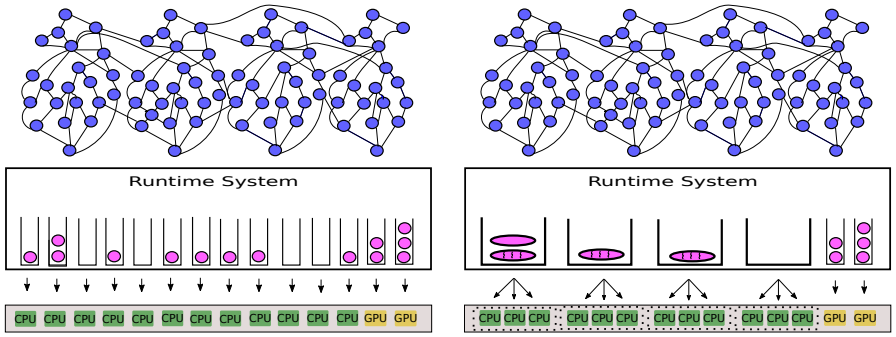
However, executing several parallel kernels simultaneously is a difficult matter because they are not aware of the resource utilization of each other and they may thus oversubscribe threads to the processing units. This issue has been first tackled within the Lithe framework [19] a resource sharing management interface that defines how threads are transferred between parallel libraries within an application. This contribution suffered from the fact that it does not allow to dynamically change the number of resources assigned to a parallel kernel. Our contribution in this study is a generalization of a previous work [16], where we introduced the so-called scheduling contexts which aim at structuring the parallelism for complex applications. Although our runtime system is able to cope with several flavors of inner parallelism (OpenMP, Pthreads, StarPU) simultaneously, in this paper we focus on the use of OpenMP to implement internal task parallelism.

3. Background

We integrate our solution to the StarPU runtime system as it provides a flexible platform to deal with heterogeneous architectures. StarPU [8] is a library that provides programmers with a portable interface for scheduling dynamic graphs of tasks onto a heterogeneous set of processing units called workers in StarPU (*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 allowing to overlap memory transfers with computations when possible.

StarPU is a platform for developing, tuning and experimenting with various task scheduling policies in a portable way. Several built-in schedulers are available, ranging from greedy and work-stealing based policies to more elaborated schedulers implementing variants of the Minimum Completion Time (MCT) policy [20]. This latter family of schedulers is based on auto-tuned history-based performance models that provide estimations of the expected lengths of tasks and data transfers. The performance model of StarPU also supports the use of regressions to cope with dynamic granularities.

4. A runtime solution to deal with nested parallelism



(a) Sequential tasks only.

(b) Parallel tasks support.

Figure 1: Managing internal parallelism within StarPU.

We introduce a set of mechanisms which aim at managing nested parallelism (i.e. task inner parallelism) within the StarPU runtime system. We consider the general case where a parallel task may be implemented on top of any runtime system. We present in Figure 1a the standard architecture of a task-based runtime system where the task-graph is provided to the runtime and the ready tasks (in purple) are dynamically scheduled on queues associated with the underlying computing resources. We propose a more flexible scheme where tasks may feature *internal parallelism* implemented using any other runtime system. This idea is represented in Figure 1b where multiple CPU devices are grouped to form *virtual resources* which will be referred to as *clusters* in the remaining of the paper. In this example, each cluster contains 3 CPU cores. We will refer to the main runtime system as the *external runtime system* while the runtime system used to implement parallel tasks will be denoted as the *inner runtime system*. The main challenges regarding this architecture are: 1) how to constrain the inner runtime system's execution to the selected set of resources, 2) how to extend the existing scheduling strategies to this new type of computing resources, and 3) how to define the number of *clusters* and their corresponding resources. In this paper, we focus on the first two problems since the latter is strongly related to online moldable/malleable task scheduling problems which are out of the scope of this paper.

Firstly, we need to aggregate cores into a cluster. This is done thanks to a simple programming interface which allows to group cores in a compact way

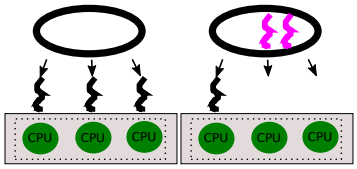
with respect to memory hierarchy. In practice, we rely on the `hwloc` framework [21], which provides the hardware topology, to build clusters containing every computing resource under a given level of the memory hierarchy (*e.g.* Socket, NUMA node, L2 cache, ...). Secondly, forcing a parallel task to run on the set of resources corresponding to a cluster depends on whether or not the inner runtime system has its own pool of threads. On the one hand, if the inner runtime system offers a multithreaded interface, that is to say the execution of the parallel task requires a call that has to be done by each thread, the inner runtime system can directly use the StarPU workers assigned to the cluster. We show in Figure 2a how we manage internal SPMD runtime systems. In this case, the parallel task is inserted in the local queue of each StarPU worker. On the other hand, if the inner runtime system features its own pool of threads (*e.g.* as most OpenMP implementations), StarPU workers corresponding to the cluster need to be paused until the end of the parallel task. This is done to avoid oversubscribing threads over the underlying resources. We describe in Figure 2b how the interaction is managed. We allow only one StarPU worker to keep running. This latter called the *master worker* of the cluster, is in charge of popping the tasks assigned to the cluster by the scheduler. When tasks have to be executed, the master worker takes the role of a regular application thread with respect to the inner runtime system. In Figure 2b, the black threads represent the StarPU workers and the pink ones the inner runtime system (*e.g.* OpenMP) threads. The master worker joins the team of inner threads while the other StarPU threads are paused.

Depending on the inner scheduling engine, the set of computing resources assigned to a cluster may be dynamically adjusted during the execution of a parallel task. This obviously 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 such as 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 cluster only at the end of the task or before starting a new one.

From a practical point of view, we integrate in a *callback* function the specific code required to force the inner runtime to run on the selected set of resources. This prologue is transparently triggered before starting executing any sequence of parallel tasks. We call this callback the *prologue callback*. This approach can be used for most inner runtime systems as the programmer can provide the implementation of the prologue callback and thus use the necessary functions in order to provide the resource allocation required for the corresponding cluster. 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. From the user point of view, provided that he has parallel implementation of his kernels, using clusters in his application is straightforward: he needs to implement the callback and create clusters. In the experimental section, we use this approach to force the `Intel MKL` library, which relies on OpenMP, to run on the set of resources corresponding to the clusters.

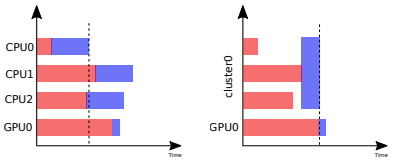
4.1. Adapting schedulers and performance models for parallel tasks

The generalization of existing greedy schedulers to handle parallel tasks is straightforward as soon as the scheduler does not rely on performance models.



(a) multi-threaded. (b) OpenMP-like.

Figure 2: Management of the pool of threads within a cluster.



(a) MCT. (b) cluster-MCT.

Figure 3: Adaptation of the MCT scheduling strategy.

Thus scheduling strategies like the work-stealing strategy can handle naturally parallel tasks. In the contrary, the MCT scheduling policy, which needs to predict the task's duration, must be adapted. Within StarPU, The task's estimated length and transfer time used for MCT decisions is computed using performance prediction models. These models are based on performance history tables dynamically built during the application execution. It is then possible for the runtime system to predict for each task the worker which will complete it at the earliest. 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.

As an illustration, we provide in Figure 3a an example showing the behavior of the MCT strategy. The blue task represents the one the scheduler is trying to assign. This task has different length on CPU and GPU devices. The choice is then made to schedule it on the CPU0 which completes it first. We have adapted the MCT strategy and the underlying performance models to be able to select a pool of CPUs when looking for a computing resource to execute a task. We have thus introduced a new type of resource: the cluster of CPUs. The associated performance model is parametrized not only by the size and type of the task together with the candidate resource but also by the number of CPUs forming the cluster. Thus, tasks can be assigned to a cluster either explicitly by the user or by the policy depending on where it would finish first. This is illustrated in Figure 3b, where the three CPUs composing our platform are grouped in a cluster. We can see that the expected length of the parallel task on the cluster is used to choose the resource having the minimum completion time for the task. Note that in this scenario, we chose to illustrate a cluster with an OpenMP-like internal runtime system.

This approach permits to deal with a heterogeneous architecture made of different types of processing units as well as clusters grouping different sets of processing units. Therefore, our approach is able to deal with multiple clusters sizes simultaneously with clusters of one CPU core and take appropriate decisions. Actually, it is helpful to think of the clusters as mini-accelerators. In this work, we let the user define sets of such clusters (mini-accelerators) and schedule tasks dynamically on top of them.

4.2. Case Study: using the parallel version of the `Intel MKL` *library*

We show in Figure 4 an example of how this tool can be used. Our aim is to 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.

```
#include <omp.h>
#include <mkl.h>

void cl_prologue()
{
  /* Get the current cluster */
  int cluster = starpu_get_current_cluster();

  /* get the CPUs of the cluster */
  int cpus[MAX_WORKERS];
  int ncpus = starpu_get_cpus(cluster, 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(...);
}
```

Figure 4: Executing `Intel MKL` parallel codes within a scheduling context.

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.

5. 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 [13, 14], which represent the computations as a DAG of tasks. We present in Figure 5 the DAG of the Cholesky factorization on a matrix containing 5x5 tiles. This figure illustrates the parallelism available with the Cholesky factorization depending on the width of the DAG.

For our evaluation, we use the Cholesky factorization of `Chameleon` [22], 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 optimized kernels from a BLAS library. Our adaptation of `Chameleon` does not change the high level task-based algorithms and subsequent DAG. We simply extend the prologue of each task to allow the use of an OpenMP implementation of `Intel MKL` (Parallel `Intel MKL`) inside the clusters and manage their creation. We call `pt-Chameleon` this adapted version of `Chameleon` which handles parallel tasks. We evaluate our approach on two different platforms: 1) an Intel Xeon Phi Knights Landing (KNL) system, 2) a modern heterogeneous multicore system equipped with GPU accelerators. The aim of this experimental study is to highlight the fact that our approach not only tackles the granularity

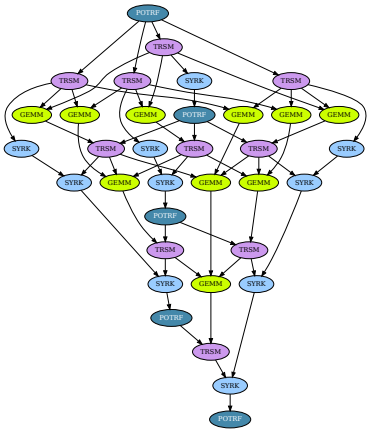


Figure 5: Task graph of a Cholesky factorization of a square matrix having five tiles on each dimension.

problem (arising on heterogeneous machines) but also provides more flexibility to efficiently exploit complex manycore systems. Although our approach is able to handle arbitrary clusters, defining the number and the size of the clusters for a given execution is a challenging problem [23]. In this paper, all experiments have been conducted using CPU clusters of the same size. We note the clusters configuration using the notation 4×16 which means we create 4 resources of 16 processing units.

5.1. Experimental evaluation on the `Intel KNL` *platform*

Our first set of experiments involves an Intel Xeon Phi-based machine, code-named KNL (Xeon Phi 7210). It is a homogeneous machine with 64 cores @1.3 GHz, each possessing 4 HyperThreads. The machine is composed of 32 tiles of 2 cores, each tile shares a 1 MB L2 cache and each core has a 32 KB L1 cache. This Xeon Phi is used as a main processor and possesses 16 GB of on-board memory, on top of 192 GB of external RAM. This platform's hardware is configurable in two ways: 1) to create locality and NUMA regions and 2) to use the on-board memory as cache or extra addressable memory. We selected the SNC-4 setting to create 4 groups of 16 cores and 4 NUMA nodes and set the on-board memory as a L3 cache. After many preliminary experiments and discussions with Intel, we found these settings to be the most efficient. We observed that the HyperThreading technology provides no performance improvement in the considered experiments and therefore we do not make use of it. Regarding the StarPU runtime system, we use the locality-aware work-stealing scheduler which was introduced in [24] and proved to be efficient for homogeneous machines.

In this section we start with a study of the performance profile of the different kernels involved in the task-based dense Cholesky factorization in order to define reasonable configurations of clusters for `pt-Chameleon` on the `Intel KNL` device for each kernel. In a second step, we evaluate these configurations in practice and provide more detailed observations on the results. Finally, we compare ourselves with two reference libraries, namely `PLASMA` and the native Parallel `Intel MKL` implementation of the dense Cholesky factorization.

	DPOTRF					DTRSM				
	480	960	1440	1920	2880	480	960	1440	1920	2880
1 core (Gflop/s)	8.818	14.695	18.646	20.417	23.515	22.837	27.554	28.614	29.157	30.226
2 cores / 1 core / 2	0.72	0.84	0.78	0.91	0.90	0.74	0.83	0.87	0.89	0.92
4 cores / 1 core / 4	0.51	0.71	0.70	0.74	0.75	0.63	0.77	0.83	0.86	0.88
8 cores / 1 core / 8	0.30	0.52	0.53	0.56	0.72	0.43	0.58	0.70	0.79	0.84
16 cores / 1 core / 16	0.19	0.31	0.39	0.42	0.52	0.30	0.44	0.55	0.62	0.69
32 cores / 1 core / 32	0.07	0.16	0.27	0.32	0.37	0.17	0.29	0.43	0.48	0.60
64 cores / 1 core / 64	0.01	0.06	0.11	0.15	0.27	0.09	0.19	0.26	0.35	0.48

(a) DPOTRF and DTRSM.

	DSYRK					DGEMM				
	480	960	1440	1920	2880	480	960	1440	1920	2880
1 core (Gflop/s)	21.384	25.725	27.207	27.905	29.400	26.703	29.339	29.872	29.911	30.856
2 cores / 1 core / 2	0.82	0.88	0.91	0.93	0.95	0.88	0.92	0.86	0.87	0.87
4 cores / 1 core / 4	0.67	0.78	0.83	0.87	0.90	0.74	0.86	0.89	0.90	0.92
8 cores / 1 core / 8	0.54	0.68	0.76	0.78	0.85	0.70	0.76	0.89	0.89	0.92
16 cores / 1 core / 16	0.39	0.53	0.62	0.69	0.74	0.46	0.71	0.82	0.86	0.90
32 cores / 1 core / 32	0.21	0.28	0.37	0.50	0.58	0.33	0.52	0.69	0.73	0.86
64 cores / 1 core / 64	0.09	0.17	0.23	0.24	0.33	0.20	0.35	0.51	0.57	0.64

(b) DSYRK and DGEMM.

Table 1: Efficiency of the four Cholesky factorization kernels when running alone on the `Intel KNL` machine with tile size 480, 960, 1440, 1920 and 2880.

We report in Table 1 the efficiency of the four kernels involved in the task-based dense Cholesky factorization on several numbers of computing cores of the `Intel KNL` device. We show these results with different input size corresponding to the typical block sizes of the task-based algorithm. These measurements were done using the `Intel MKL` kernels which we run alone on the machine. First of all, we notice that for all kernels, the more we increase the input data size the more the sequential kernel is efficient. This behavior is unusual: the peak performance of MKL kernels is typically reached with moderate data sizes (e.g. 320 or 480) on regular Xeon cores. Secondly, the efficiency of the parallel kernels is limited for large numbers of processor counts mainly due to the limited size of the input matrix. Moreover, for medium numbers of cores (i.e. 8 and 16), the efficiency is satisfactory for large tile sizes of 1920 or above, and acceptable for medium ones of 960 and 1440. Finally, some kernels, namely DGEMM and DSYRK, are more efficient and scalable than others, namely DPOTRF and DTRSM. These results illustrate that processing large tiles over medium sets of processors is probably a good way to perform efficient Cholesky factorizations on `Intel KNL` processors. On the other hand, relying on small tile sizes on small number of cores may also exhibit good performance. We will come back on these observations later in this paper to discuss the performance of the Cholesky factorization operation of the `pt-Chameleon` implementation on the `Intel KNL` platform. Finally, we performed the same measurements in a context where the platform was loaded (i.e. several kernels were running concurrently on different processing units) and we observed that the kernels are slower due to memory contention. As an illustration, if we consider the DGEMM kernel with a matrix order of 480 for the sequential task case, the performance reduction is of 15% due to the created noise, whereas for the case with 16 threads and a matrix order of 2880, the performance reduction is of 4%.

We present in Figure 6 the experimental evaluation of the `Chameleon` and `pt-Chameleon` behavior with the Cholesky factorization for different tile sizes

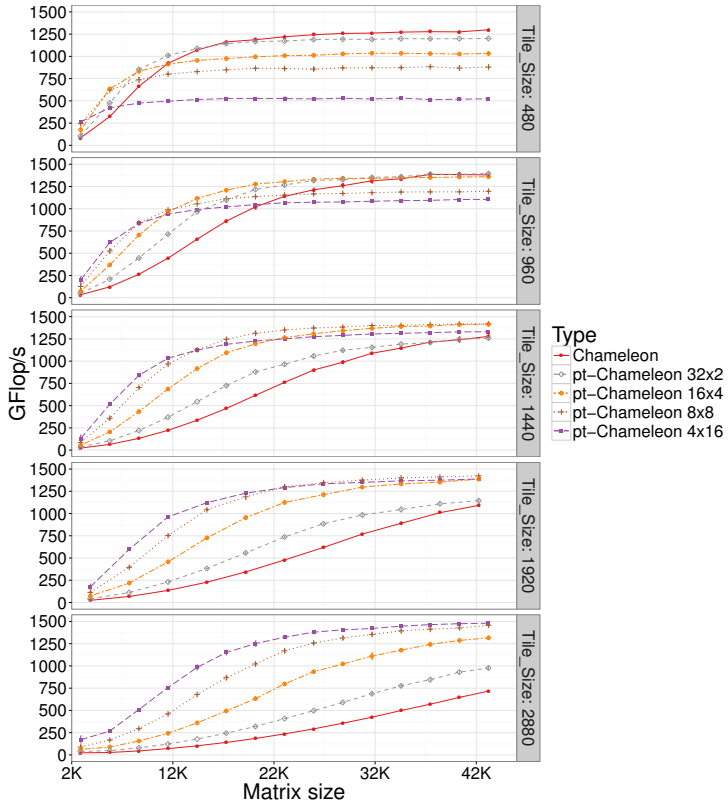


Figure 6: Performance of the `pt-Chameleon` Cholesky factorization for different clusters configurations and different tile sizes. 32×2 and 16×4 configurations will be referred to as small configurations. 8×8 and 4×16 configuration will be referred to as intermediate configurations. 2×32 will be referred to as large configuration.

and matrix sizes. This figure confirms that, for small tile sizes, exploiting internal task parallelism is not effective (as already observed in Table 1). In this case, `Chameleon` and `pt-Chameleon` using 16 clusters of 4 cores each are the most efficient versions. However, to rely on efficient kernels one should use larger tile size (see the absolute performance of the kernels according to the tile size in Table 1). With the increase of the tile size, we can see that configurations relying on larger clusters, namely 8 (resp. 4) clusters of 8 (resp. 16) cores each, become increasingly efficient while the performance of `Chameleon` drops. The loss of performance of `Chameleon` can be explained by the fact that when increasing the tile size, the amount of task parallelism is strongly reduced leading to larger idle times on the computing resources (especially at the beginning and at the end of the execution). Additionally, we can observe that for `pt-Chameleon` with very large tile size (2880), the peak performance is reached for the configuration consisting of 4 clusters with 16 cores each, reaching 1495.62 Gflop/s. This illustrates the interest of relying on parallel tasks and clusters to find a better tradeoff between kernel efficiency and the degree of parallelism in the task graph. Moreover, for moderate matrix orders, i.e. lower than 12K, the versions relying on large granularity and large clusters configurations are less

efficient than the ones with fine grain tasks and small clusters. This is mainly explained by the fact that relying on large tile sizes strongly reduces both the amount of tasks and task parallelism leading to poor performance for moderate size problems. However, for these matrix sizes, one can rely on intermediate configurations (8 clusters with 8 cores or 16 clusters with 4 cores each to be more precise) and tile size (480 or 960) to obtain a performance gain of up to 80% over `Chameleon`.

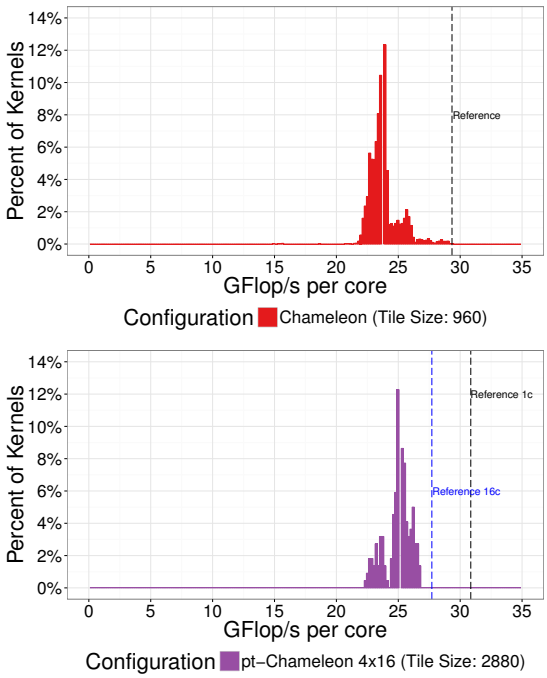


Figure 7: Comparison of the DGEMM kernel behavior with the best configurations for both `Chameleon` and `pt-Chameleon` with a matrix order of 34560.

We provide in Figure 7 a more detailed comparison of `Chameleon` and `pt-Chameleon` through the measurement of the actual DGEMM kernels performance, DGEMM being the dominant kernel for large matrix sizes. For this figure, we measured the performance of each individual task of the Cholesky factorization for all our configurations. For the sake of clarity, we only present results for a large matrix size (matrix of order 34560) using the best configurations coming from Figure 6 for both `Chameleon` (tile size of 960) and `pt-Chameleon` (tile size of 2880 with 4 clusters of 16 cores each). We note that with this setup for both the `Chameleon` and `pt-Chameleon` configurations the idleness is under 0.1%, therefore showing that the parallelism is well adapted to the machine in both cases and do not come into account in this comparison. The results in Figure 7 are presented in an histogram plot presenting the percentage of DGEMM tasks running at a given performance (Gflop/s). Note that in order to ease the comparison between `Chameleon` and `pt-Chameleon`, the results for the latter were brought to one core. Moreover, we also provide the so-called reference performance of the kernel (vertical dashed line) which represents its performance when run-

ning alone on the device as provided in Table 1. We can observe on this figure that the average performance of the DGEMM kernels for the `pt-Chameleon` version is higher by a factor of 5.1% than the average performance of the DGEMM kernels for `Chameleon`. Furthermore, we can see that the performance of the `pt-Chameleon` version is much more stable than the one of `Chameleon` and is also much closer on average to its reference performance. These observations explain why `pt-Chameleon` outperforms `Chameleon` on the `Intel KNL` platform.

We compare in Figure 8 the best configurations among the ones presented previously with the native implementation of the Cholesky factorization from the `Intel MKL` for the `Intel KNL` platform and with the `PLASMA` library. It is important to note that we were not able to compare our approach with the `MAGMA` library for `Intel KNL` architectures (see [25] for more details) because the corresponding software package is not yet available. For the sake of clarity, we only report the results obtained with the best setup for each library. Both `Chameleon` and `PLASMA` reached their peak performance when used with a tile size of 480. Concerning `Intel MKL` we simply call the DPOTRF subroutine on the whole matrix. Finally, for `pt-Chameleon`, we present the results when using: 1) a tile size of 960 with 8 clusters having 8 cores each, 2) a tile size of 2880 with 4 clusters having 16 cores each.

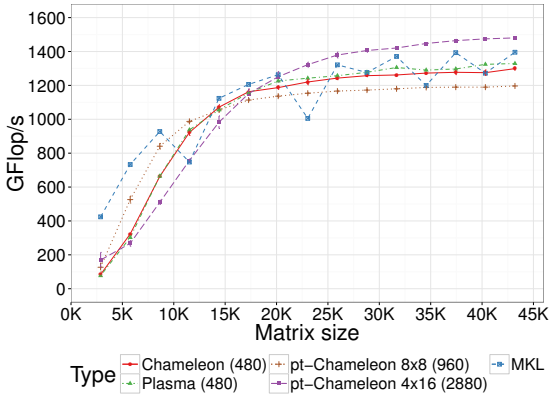


Figure 8: Performance comparison of the Cholesky factorization on `Intel KNL` with `pt-Chameleon`, `Chameleon`, `PLASMA` and `Intel MKL`.

We can see that `Chameleon` and `PLASMA` have a very similar behavior: the obtained performance is reasonable for small to intermediate matrices while they get outperformed for the large ones. Concerning the reference curve represented by `Intel MKL`, we can see that it outperforms all the other implementations for small problems. However for intermediate to large problems, the `pt-Chameleon` implementation with large tile size has higher peak performance and smoother performances (we are not able right now to explain all the glitches on the `Intel MKL` curve). `pt-Chameleon` achieves a peak performance of 1495.62 Gflop/s with a 2880 tile size for large matrices which represents an absolute improvement of 7.2% over the `Intel MKL` performance. Concerning the `pt-Chameleon` configuration using 8 clusters with 8 cores each, we can see that its absolute performance is low. However, for small to medium matrix sizes, it provides the closest performance to the one of `Intel MKL`.

These results illustrate the interest of studying tradeoffs between inter-task and internal parallelism on many-core processors. This approach allows to rely on coarse grained tasks when needed without suffering from the drastic decrease to the amount of parallelism. Moreover, the results highlight how StarPU succeeds in handling the parallel `Intel MKL` kernels thanks to the clusters introduced in this paper. From a software point of view, `pt-Chameleon` could switch from one configuration to another depending on the size of the input matrix providing thus a very efficient and portable implementation of the dense Cholesky factorization operation.

5.2. Experimental evaluation on modern heterogeneous systems

For the following set of experiments, the machine we use is heterogeneous and composed of 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. In StarPU one core is dedicated to each GPU, consequently we report on all figures performance with 20 cores for the `Chameleon` and `pt-Chameleon` versions. We used a configuration for `pt-Chameleon` composed of 2 clusters aggregating 10 cores each (noted 2×10), so that the 10 cores of a CPU belong to a single cluster. In comparison, `Chameleon` uses 20 sequential CPU cores on this platform. Regarding the StarPU runtime system, we use for these experiments the adapted version of the MCT scheduler which was introduced in Section 4.1. Finally, we show on all figures the average performance and observed variation over 5 runs on square matrices.

	DPOTRF		DTRSM		DSYRK		DGEMM	
	960	1920	960	1920	960	1920	960	1920
1 core (Gflop/s)	27.78	31.11	34.42	34.96	31.52	32.93	36.46	37.27
GPU / 1 core	1.72	5.95	8.72	18.59	26.96	31.73	28.80	30.86
10 cores / 1 core	5.55	7.48	6.75	8.48	6.90	8.63	7.77	8.56

Table 2: Acceleration factor of Cholesky factorization kernels on a GPU and 10 cores compared to one core with tile size 960 and 1920.

We report in Table 2 the acceleration factors of using 10 cores or one GPU compared to the single core performance for each kernel of the Cholesky factorization. We conduct our evaluation using `Intel MKL` for the CPUs and CuBLAS (resp. MAGMA) for the GPUs. This table highlights a sublinear scalability of using 10 cores compared to using 1 core. On our best kernel DGEMM we accelerate the execution by a factor of 7.77 when using 10 cores and this increases to 8.56 with a tile size of 1920. Despite this, we can see that relying on sequential kernels worsens the performance gap between the CPUs and GPUs while relying on clusters makes the set of computing resources more homogeneous. We can obtain an acceleration factor of GPU against CPUs by dividing the second line by the third one. For example, the performance gap for the DGEMM kernel with a tile size of 960 is 28.8 when using 1 core compared to a GPU whereas it is $28.80/7.77 \simeq 3.7$ when using 10 cores compared to a GPU. As a consequence, if 28 independent DGEMM of size 960 are submitted on computer of 10 cores and a GPU, the `Chameleon` scheduler assigns all the tasks to the GPU whereas `pt-Chameleon` assigns 6 tasks to the 10 core cluster and 22 tasks to GPUs. Another important aspect which can compensate some loss in efficiency

is the `pt-Chameleon` ability to accelerate the critical path. Indeed, a cluster of 10 cores can execute the DPOTRF kernel on a tile size of 960 three times faster than on a GPU. The performance is also almost the same for the DTRSM task.

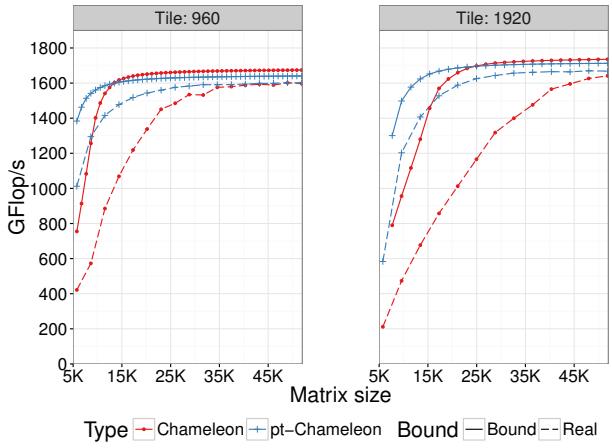


Figure 9: Comparison of the `pt-Chameleon` and `Chameleon` Cholesky factorization with computed bounds. 20 CPUs and 1 GPU are used.

We show in Figure 9 the performance of the Cholesky factorization for both `Chameleon` and `pt-Chameleon` with multiple tile sizes and their computed makespan theoretical lower bounds. These bounds are computed thanks to the iterative bound technique introduced in [26] which iteratively adds new critical paths until all are taken into account. As these bounds do not take communications with GPU devices into account, they are clearly unreachable in practice. These bounds show that `pt-Chameleon` can theoretically obtain better performance than `Chameleon` on small to medium sized matrices. Indeed, the CPUs are underutilized in the sequential tasks case due to a lack of parallelism whereas using clusters lowers the amount of tasks required to feed the CPU cores. The 5K matrix order point shows a difference of performance of 600 Gflop/s, this is close to the obtainable performance on these CPUs. For both tile sizes on large matrices (*e.g.* 40K), the `Chameleon` bound is over the `pt-Chameleon` one. This is due to the better efficiency of the sequential kernels since the parallel kernels do not exhibit perfect scalability, allowing the CPUs to achieve better performance per core in the sequential case. We observe that for a coarser kernel grain of 1920, the maximum achievable performance is higher, mainly thanks to a better kernel efficiency on GPUs with this size. For DGEMM kernel we can gain close to 100 Gflop/s (or 10%). We can also note that the gap between `Chameleon` and `pt-Chameleon` bound decreases slightly as we increase the tile size to 1920 thanks to a relatively better gain in efficiency per core compared to the sequential one. Additionally, the real executions are underneath the theoretical bounds. This is due to the fact that transfer time is not taken into account in the bounds. Moreover, the online MCT scheduler can exaggeratedly favor GPUs because of their huge performance bonus in the `Chameleon` case as was shown in [26]. Finally, this figure highlights a constantly superior performance of `pt-Chameleon` over `Chameleon` which achieves up to 65% better

performance on a matrix size of 11K for the 960 tile size case and up to 100% better performance on matrices lower than 10K. On those matrix sizes, real `pt-Chameleon` execution is able to go over the theoretical bound of `Chameleon` which demonstrates the superiority of our approach.

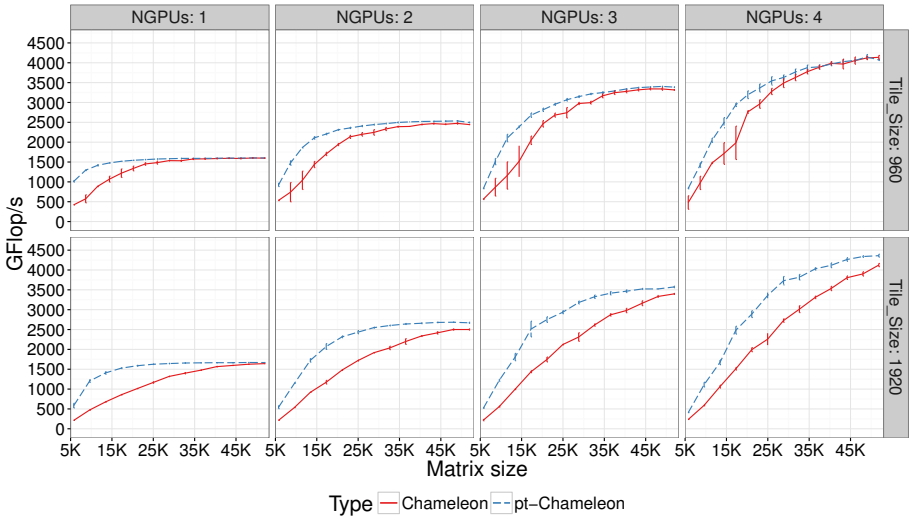


Figure 10: Performance of the Cholesky factorization with `pt-Chameleon` and `Chameleon` with varying number of GPUs and task granularity.

We report in Figure 10 the performance behavior of our implementation of the Cholesky factorization using the `pt-Chameleon` framework, compared to the existing `Chameleon` library. When looking at medium sized matrices we observe that `pt-Chameleon` is able to achieve significantly higher performance than `Chameleon` across all test cases. On those matrices, the `Chameleon` library has some performance variability. This is mainly due to bad scheduling decisions regarding tasks on the critical path in `Chameleon`. Indeed, if an important task is wrongly scheduled on a CPU such as a DPOTRF, we may lack parallelism for a significant period of time. Whereas in the `pt-Chameleon` case using parallel tasks even accelerates the critical path due to a better kernel performance, which makes the approach less sensitive to bad scheduling decisions, lowering `pt-Chameleon`'s variance. Both `Chameleon` and `pt-Chameleon` showcase a good scalability when increasing the number of GPUs. For example the peak for 1 GPU with a tile size of 960 is at 1.6 Tflop/s and for 2 GPUs it goes up to 2.6 Tflop/s. This improvement is as expected since 1 Tflop/s is the performance of a GPU on this platform with the DGEMM kernel. `Chameleon` scales slightly less than `pt-Chameleon` with a coarse task grain size of 1920. The gap between the two versions increases when increasing the number of GPUs. As shown previously, the scheduler can schedule too many tasks on the GPUs leading to a CPU under-utilization with such a high factor of heterogeneity.

Another factor is the cache behavior of both implementations. Indeed, each processor benefits of 30MB cache and by using one cluster per processor instead of 10 independent processing units we lower by 10 the working set size. Since a tile of 960 weights 7MB whereas a tile of 1920 weights 28MB we are even

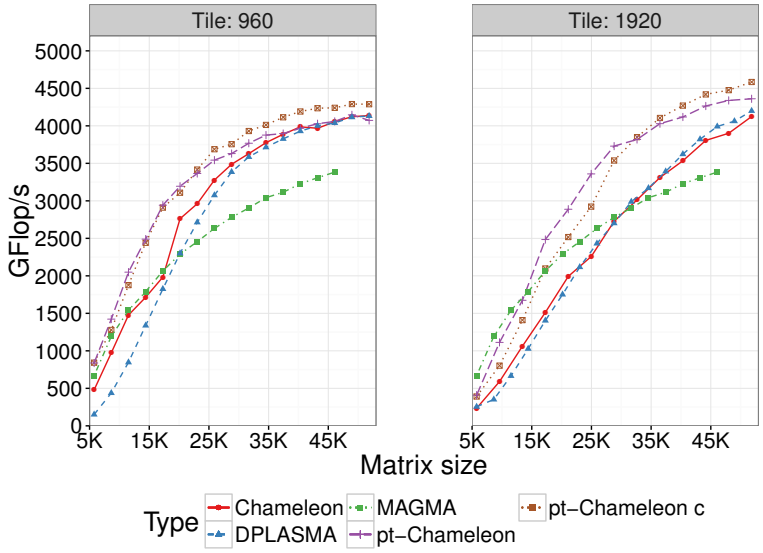


Figure 11: Comparison of the constrained `pt-Chameleon` with baseline `Chameleon`, `MAGMA` (default parameters and multithreaded `Intel MKL`) and hierarchical `DPLASMA` (internal blocking of 192 (left) and 320 (right)).

able to fit entirely a 1920 tile in the LLC. This highlights another constraint: the memory contention bottleneck. We had to explicitly use the *numactl* tool to allocate pages in a round robin fashion on all 4 NUMA nodes, otherwise the behavior of `Chameleon` was very irregular. In fact, even with the interleave setting, we observed that some compute intensive kernels such as DGEMM could become more memory bound for the `Chameleon` case with a matrix size of 43K. To investigate this issue we conducted an experiment using Intel VTune where we allocated the complete matrix on one NUMA node thanks to the numactl tool. We saw that for `Chameleon` 59% of the DGEMM kernels were bounded by memory, whereas for `pt-Chameleon` only 13% were bounded by memory. We also observed over two times less cache misses on our `pt-Chameleon` version.

Finally, in Figure 11 we compare `pt-Chameleon` to multiple dense linear algebra reference libraries: `MAGMA`, `Chameleon` and `DPLASMA` using the hierarchical granularity scheme presented in [15]. We make use of a constrained version (2×10c) where the DPOTRF and DTRSM tasks are restricted to CPU workers. On this figure, the `MAGMA` and `DPLASMA` versions use the 24 CPU cores. This strategy is comparable to what is done in [15] where only DGEMM kernels are executed on GPU devices. We observe that using the regular MCT scheduler for small matrices leads to better performance since in the constrained version the amount of work done by CPUs is too large. However, when we increase the matrix size, the constrained version starts to be efficient and leads to a 5% increase in performance on average, achieving a peak of 4.6 Tflop/s on our test platform. We see that the absolute peak is obtained by `pt-Chameleon` which outperforms all the other implementations.

These results highlight the portability both in terms of software and performance of the approach as it is able to tackle modern heterogeneous systems

transparently. Our approach also allows us to tackle the granularity problems arising with such heterogeneous systems by reducing the amount of resources and the gap between CPU and GPU resources performance. This allows our approach to reach higher performance by changing the task granularity for large matrix sizes and obtaining very better performance for the low matrix sizes due to the lower parallelism requirement.

6. Conclusion

One of the biggest challenge raised by the development of high performance task-based applications on top of heterogeneous hardware lies in coping with the increasing performance gap between accelerators and individual cores. One way to address this issue is to use multiple tasks' granularities, but it requires in-depth modifications to the data layout used by existing implementations.

We propose a less intrusive and more generic approach that consists in reducing the performance gap between processing units by forming clusters of CPUs on top of which we exploit tasks' inner parallelism. Performance of these clusters of CPUs can better compete with the one of powerful accelerators such as GPUs. Our implementation extends the StarPU runtime system so that the scheduler only sees virtual computing resources on which it can schedule parallel tasks (*e.g.* BLAS kernels). The implementation of tasks inside such clusters can virtually rely on any thread-based runtime system, and runs under the supervision of the main StarPU scheduler.

We demonstrate the relevance of our approach using task-based implementations of the dense linear algebra Cholesky factorization. We show that for modern heterogeneous machines our implementation is able to outperform the `MAGMA`, `DPLASMA` and `Chameleon` state-of-the-art dense linear algebra libraries while using the same task granularity on accelerators and clusters.

In addition, we show that the proposed approach not only tackles the granularity problem but also offers more flexibility to efficiently exploit modern many-core systems with complex memory hierarchies and locality requirements. The study conducted in this paper shows that this same implementation is able to efficiently exploit the `Intel KNL` device while outperforming both `Chameleon`, `Intel MKL` and `PLASMA`. This illustrates the performance portability of our approach on modern high-performance computing systems.

In the near future, we intend to further extend this work by investigating how to automatically determine the optimal size of clusters. Preliminary experiments show that using clusters of different sizes sometimes leads to significant performance gains [23]. Thus, we plan to design heuristics that could dynamically adapt the number and the size of clusters on the fly, based on statistical information regarding ready tasks. Finally, we plan also to study how internal parallelism within tasks can be used in the context of more complex applications like sparse linear solvers or fast multipole methods.

Acknowledgment. We are grateful to Mathieu Faverge for his help for the comparison of `DPLASMA` and `pt-Chameleon`. We are also grateful to Mikko Byckling for his help and comments on the `Intel KNL` setup. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed.

7. bibliography

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

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

[3] T. D. R. Hartley, E. Saule, Ü. V. Çatalyürek, Improving performance of adaptive component-based dataflow middleware, Parallel Computing 38 (6-7) (2012) 289–309.

[4] D. M. Kunzman, L. V. Kalé, Programming heterogeneous clusters with accelerators using object-based programming, Scientific Programming 19 (1) (2011) 47–62.

[5] E. Hermann, B. Raffin, F. Faure, T. Gautier, J. Allard, Multi-gpu and multi-cpu parallelization for interactive physics simulations, in: Euro-Par 2010 - Parallel Processing, Vol. 6272, 2010, pp. 235–246.

[6] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: expressing locality and independence with logical regions, in: SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012, 2012, p. 66.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J. J. Dongarra, PaRSEC: Exploiting heterogeneity to enhance scalability, Computing in Science and Engineering 15 (6) (2013) 36–45.

[8] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, Concurr. Comput. : Pract. Exper. 23 (2011) 187–198.

[9] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, E. Quintana-Ortí, An Extension of the StarSs Programming Model for Platforms with Multiple GPUs, in: Euro-Par 2009, 2009, pp. 851–862.

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

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

[12] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. V. Zee, E. Chan, Programming matrix algorithms-by-blocks for thread-level parallelism, ACM Trans. Math. Softw. 36 (3).

[13] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, Journal of Physics: Conference Series 180 (1).

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

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

[16] A. Hugo, A. Guermouche, P. Wacrenier, R. Namyst, Composing multiple starpu applications over heterogeneous machines: A supervised approach, IJHPCA 28 (3) (2014) 285–300.

[17] F. Song, S. Tomov, J. Dongarra, Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems, in: Proceedings of ICS'12, 2012, pp. 365–376.

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

[19] H. Pan, B. Hindman, K. Asanović, Composing parallel software efficiently with lithe, SIGPLAN Not. 45 (2010) 376–387.

[20] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 260–274.

[21] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: A generic framework for managing hardware affinities in HPC applications, in: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 180–186.

[22] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, A hybridization methodology for high-performance linear algebra software for gpus, GPU Computing Gems, Jade Edition 2 (2011) 473–484.

[23] O. Beaumont, T. Cojean, L. Eyraud-Dubois, A. Guermouche, S. Kumar, Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources, in: International Conference on High Performance Computing, Data, and Analytics (HiPC 2016), Hyderabad, India, 2016.

[24] E. Agullo, A. Buttari, A. Guermouche, F. Lopez, Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems, ACM Trans. Math. Softw. 43 (2).

[25] A. Haidar, S. Tomov, K. Arturov, M. Guney, S. Story, J. Dongarra, Lu, qr, and cholesky factorizations: Programming model, performance analysis and optimization techniques for the Intel Knights Landing Xeon Phi, in: IEEE High Performance Extreme Computing Conference (HPEC'16), 2016, p. to appear.

[26] E. Agullo, O. Beaumont, L. Eyraud-Dubois, S. Kumar, Are Static Schedules so Bad ? A Case Study on Cholesky Factorization, in: Proceedings of IPDPS'16, 2016.