Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms

Emmanuel Agullo
INRIA Bordeaux – Sud-Ouest
Talence, France
Email: emmanuel.agullo@inria.fr

Olivier Beaumont
INRIA Bordeaux – Sud-Ouest
Institut de Mathématiques de Bordeaux
Talence, France
Email: olivier.beaumont@inria.fr

Lionel Eyraud-Dubois
INRIA Bordeaux – Sud-Ouest
Talence, France
Email: lionel.eyraud-dubois@labri.fr

Julien Herrmann
Ecole Normale Supérieure de Lyon
Lyon, France
Email: julien.herrmann@ens-lyon.fr

Suraj Kumar
INRIA Bordeaux – Sud-Ouest
Talence, France
Email: suraj.kumar@inria.fr

Loris Marchal
CNRS and University of Lyon
Lyon, France
Email: loris.marchal@ens-lyon.fr

Samuel Thibault
University of Bordeaux,
INRIA Bordeaux – Sud-Ouest
Talence, France
Email: samuel.thibault@labri.fr

***Abstract*—We consider the problem of allocating and scheduling dense linear application on fully heterogeneous platforms made of CPUs and GPUs. More specifically, we focus on the Cholesky factorization since it exhibits the main features of such problems. Indeed, the relative performance of CPU and GPU highly depends on the sub-routine: GPUs are for instance much more efficient to process regular kernels such as matrix-matrix multiplications rather than more irregular kernels such as matrix factorization. In this context, one solution consists in relying on dynamic scheduling and resource allocation mechanisms such as the ones provided by PaRSEC or StarPU. In this paper we analyze the performance of dynamic schedulers based on both actual executions and simulations, and we investigate how adding static rules based on an offline analysis of the problem to their decision process can indeed improve their performance, up to reaching some improved theoretical performance bounds which we introduce.**

***Keywords*-Cholesky Factorization – Dense Linear Algebra – Simulation – Heterogeneous Resources – Scheduling – Resource Allocation – Dynamic Schedulers.**

I. Introduction

Linear algebra operations are the basis of many scientific operations. Our objective is to optimize the performance of one of them (namely the Cholesky decomposition) on a hybrid computing platform. The use of GPUs and other accelerators such as Xeon Phi are common ways to increase the computation power of computers at a limited cost. The large computing power available on such accelerators for regular computation makes them unavoidable for linear algebra operations. However, optimizing the performance of a complex computation on such a hybrid platform is very complex, and a manual optimization seems out of reach given the wide variety of hybrid configurations. Thus, several runtime systems have been proposed to dynamically schedule a computation on hybrid platforms, by mapping parts of the computation to each processing elements, either cores or accelerators. Among other successful projects, we may cite StarPU [1] from INRIA Bordeaux (France), QUARK [2] and PaRSEC [3] from ICL, Univ. of Tennessee Knoxville (USA), Supermatrix [4] from University of Texas (USA), StarSs [5] from Barcelona Supercomputing Center (Spain) or KAAPI [6] from INRIA Grenoble (France). Usually, the structure of the computation has to be described as a task graph, where vertices represent tasks and edges represent dependencies between them. Most of these tools enable, up to a certain extent, to schedule an application described as a task graph onto a parallel platform, by mapping individual tasks onto computing resources and by performing data movements between memories when needed.

There is an abundant literature on the problem of scheduling task graphs on parallel processors. This problem is known to be NP-complete [7]. Lower-bounds based either on the length of the critical path (the longest path from an entry vertex to an output vertex) or on the overall workload (assuming ideal parallelism) have been proposed, and simple list-scheduling algorithms are known to provide $2 - 1/m$-approximation on homogeneous platforms, at least when communication times are negligible [8]. Several scheduling heuristics have also been proposed, and among them the best-known certainly is heterogeneous early finish time (HEFT) [9], which inspired some dynamic scheduling

strategies used in the above-mentioned runtimes. However, it remains a large gap between the theoretical lower-bounds and the actual performance of dynamic HEFT-like heuristics. Another way to assess the quality of a scheduling strategy is to compare the actual performance to the machine peak performance of the computing platform computed as the sum of the performance of its individual computational units. Rather than this machine peak performance which is known to be unreachable, one usually considers the GEMM peak obtained by running matrix multiplication kernels (GEMMs). For large matrices, the task-graph of a Cholesky factorization exhibits a sufficient amount of parallelism, and a sufficient number of GEMM calls for this bound to be reasonable. However, on small and medium size matrices, there is still a large gap between GEMM peak performance and the best-achievable Cholesky performance.

In this paper, we optimize the dynamic scheduling of the Cholesky decomposition of a dense, symmetric, and positive-definite double-precision matrix A, into the product LL^T, where L is lower triangular and has positive diagonal elements, using one runtime system, StarPU, and provide better makespan bounds to prove the quality of our schedules. The contributions of the paper are:

- Better lower bounds on the makespan of a Cholesky factorization on a parallel hybrid platform;
- Better dynamic schedules, based not only on HEFT but also on an hybridization of static and dynamic task assignments;
- A very efficient schedule for a simple hybrid platform model, achieved by constraint programming.
- Numerous experiments to assess the performance of our schedules using the StarPU runtime.

Note that what is done here using StarPU could have been done with other runtimes, provided that we are able to control their mapping and scheduling policies. Similarly, we could have chosen another dense linear algebra factorization such as the QR or LU decompositions.

II. Context

A. Cholesky factorization

The Cholesky factorization (or Cholesky decomposition) is mainly used to solve a system of linear equations $Ax = b$, where A is a $N \times N$ symmetric positive-definite matrix, b is a vector, and x is the unknown solution vector to be computed. Such systems often arise in physics applications, especially when looking for numerical solutions of partial differential equations, where A is positive-definite due to the nature of the modeled physical phenomenon. One way to solve such a linear system is first to compute the Cholesky factorization $A = LL^T$, where L (referred to as the Cholesky factor) is a $N \times N$ real lower triangular matrix with positive diagonal elements. The solution vector x can then be computed by solving the two following triangular systems: $Ly = b$ and $L^T x = y$.

Algorithm 1: Pseudocode of the tiled Cholesky factorization

```
for k = 0 to n − 1 do
    A[k][k] ← POTRF(A[k][k]);
    for i = k + 1 to n − 1 do
        A[i][k] ← TRSM(A[k][k], A[i][k]);
    for j = k + 1 to n − 1 do
        A[j][j] ← SYRK(A[j][k], A[j][j]);
        for i = j + 1 to n − 1 do
            A[i][j] ← GEMM(A[i][k], A[j][k], A[i][j]);
```

To take advantage of modern highly parallel architectures, state-of-the-art numerical algebra libraries implement tiled Cholesky factorizations. The matrix $A = (A_{ij})_{0 \leq i,j \leq n}$ is divided into $n \times n$ tiles (or blocks) of $n_b \times n_b$ elements, and the tiled Cholesky algorithm can then be seen as a sequence of tasks that operate on small portions of the matrix. This approach greatly improves the parallelism of the algorithm and mostly involves BLAS3 kernels whose library implementations are really fast on modern architectures. The benefits of such an approach on parallel multicore systems have already been discussed in the past [10], [11], [12]. Following the BLAS and LAPACK terminology, the tiled algorithm for Cholesky factorization is based on the following set of kernel subroutines:

- *POTRF:* This LAPACK subroutine is used to perform the Cholesky factorization of a symmetric positive definite tile A_{kk} of size $n_b \times n_b$ producing a lower triangular tile L_{kk} of size $n_b \times n_b$.
- *TRSM:* This BLAS subroutine is used to apply the transformation computed by *POTRF* to a A_{ik} tile by means of a triangular system solving.
- *GEMM:* This BLAS subroutine computes a matrix-matrix multiplication of two tiles A_{ik}, A_{jk} and subtract the result to the tile A_{ij}. The old value of A_{ij} is overwritten by the new one.
- *SYRK:* This BLAS subroutine executes a symmetric rank-k update on a diagonal tile A_{kk}.

Note that no extra memory area is needed to store the L_{ij} tiles since they can overwrite the corresponding A_{ij} tiles from the original matrix. The tiled Cholesky algorithm can be written as in Algorithm 1.

In this sequential pseudocode, we can notice that some kernel subroutines depend on each other, while others can be processed in parallel. Such an algorithm is commonly represented by its task graph (or DAG) that depicts its actual dependencies. In this well established model, each vertex of the graph represents a call to one of the four kernel subroutines presented above. The edges between two task vertices represent a direct data dependency between

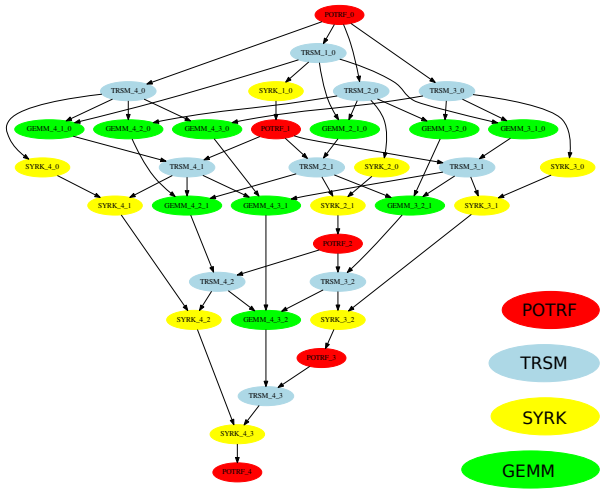


Figure 1. Task graph for the Cholesky decomposition of a 5×5 tiled matrix.

tasks. Figure 1 depicts the task graph for the Cholesky decomposition of a 5×5 tiled matrix.

B. Multiprocessor scheduling

1) Static task allocation: It is well known that the allocation of the tasks to the computing cores affect the performance and scalability, because of data locality and task heterogeneity. This problem has been adressed in the distributed memory context. For example, the ScaLAPACK library [13] first distributes the matrix tiles to the processors, using a standard 2D block-cyclic distribution of tiles along a virtual p-by-q homogeneous grid. In this layout the p-by-q top-left tiles of the matrix are topologically mapped onto the processor grid and the rest of the tiles are distributed onto the processors in a round-robin manner. It then implements an owner-compute strategy for task allocation: a task overwriting a tile is executed on the processor hosting this tile. This layout is also incorporated in the High Performance Fortran standard [14]. It ensures a good load and memory usage balancing for homogeneous computing resources [13]. However, for heterogeneous resources, this layout is no longer an option, and dynamic scheduling is a widespread practice.

These ideas also make sense in a shared-memory environment in order to take advantage of data locality. For instance the PLASMA [15] library provides an option for relying on such static schedules on multicore chips.

2) Dynamic task graph scheduling: Dynamic strategies have been developed in order to design methods flexible enough to cope with unpredictable performance of resources, especially in the context of real time systems, where on-line and adaptive scheduling strategies are required [16], [17]. More recently, the design of dynamic schedulers received a lot of attention, since on modern heterogeneous and possibly shared systems, the actual prediction of either execution and communication times is very hard, thus justifying the design of ad-hoc tools that will be described in Section IV.

As presented earlier, many scheduling heuristics have been proposed for DAGs since this problem is NP-complete. Most of these heuristics are list-scheduling heuristics: they sort tasks according to some criterion and then schedule them greedily. This makes them good candidates to be turned into dynamic scheduling heuristics. The best-known list-scheduling heuristic for DAGs on heterogeneous platforms is certainly HEFT [9]. It consists in sorting tasks by decreasing *bottom-level*, which is the weight of the longest path from a task to an exit task (a task without successors). In a heterogeneous environment, the weight of a task (or communication) is computed as the average computation (or communication) time over the whole platform. Then, each task is considered and scheduled on the resource on which it will finish the earliest. HEFT turns out to be an efficient heuristic for heterogeneous processors. Other approaches have been proposed to avoid data movement when taking communications into account, such as clustering tasks into larger granularity tasks before scheduling them [18].

III. Makespan lower bounds

Performance results for linear algebra computation are often accompanied with an upper bound in terms of FLOP/s (FLoating-point Operations Per Second), in order to assess the achieved efficiency. Since the theoretical peak performance is usually unreachable, particularly with GPUs, the common bound being used is the performance of a simple matrix multiplication (GEMM) since this is the most efficient dense linear algebra operation, and thus providing a good hint of some achievable performance. This bound takes into account the heterogeneity of the platform by summing up the obtained GFLOP/s (GigaFLOP/s) on the various processing elements. It however does not take into account the *heterogeneity of the application*, which is particularly important for small and medium matrices, for which a fair amount of the tasks are not GEMMs but much less efficient tasks such as POTRFs, especially on accelerators.

We here propose much more accurate bounds that take into account both heterogeneity of the computation resources and of the application kernels, by taking as input the execution time of any kernel on any type of resource. They also to a certain extent take into account the task graph itself, in terms of task dependencies.

A. Linear Programming formulation

The makespan lower bound computation is based on a relaxation of the scheduling problem, in which almost all precedence constraints are ignored. This formulation focuses on the number of tasks n_{rt} of each type t (GEMM, SYRK, TRSM, POTRF) which are executed on each resource type r

(CPU, GPU, ...). From the Cholesky task graph, we know the number N_t of tasks of each type t that need to be performed on the whole platform, and from the platform we know the number M_r of processing elements of each type r available to schedule the tasks. For each task type t and resource type r, the calibration mechanisms inside StarPU (described in Section IV-A) provide the execution time T_{rt} of these tasks on this resource type. The basic area bound is obtained by solving the following linear problem:

$$
\begin{aligned}
& \text{minimize the makespan } l \text{ such that} \\
\forall t, \quad & \text{all } N_t \text{ tasks of type } t \text{ get executed over the various} \\
& \text{processing element types } r\text{:} \\
& \sum_r n_{rt} = N_t \\
\forall r, \quad & \text{the } M_r \text{ resources of type } r \text{ complete all their tasks} \\
& \text{of various types } t \text{ within the makespan } l\text{:} \\
& \sum_t n_{rt} T_{rt} \leq l \times M_r \\
\forall r, t \quad & n_{rt} \in \mathbb{N}^+
\end{aligned}
$$

It is clear that the optimal value l^* of this linear program is a lower bound on the total execution time of the task graph, since any execution needs to execute all tasks. Ignoring the task graph precedences in this bound allows one to handle tasks of the same type with a couple of variables (one per resource type), instead of having one variable for each task in the graph, thus limiting the number of variables and reducing symmetries in the solution space. While being very naive, this formulation allows StarPU, without any input from the application beyond the normal task submission, to automatically generate it and solve it on the fly very quickly, right after the application execution, which thus allows one to print this theoretical bound along the measured performance in the application output.

Due to the actual timings of the different task types, this linear program always decides that all POTRF tasks should be executed on CPUs, since all other task types make much more efficient use of the GPU resources. However, in practice all POTRF tasks are on the critical path of the Cholesky graph, and hence this implies that the resulting lower bound is too optimistic for small matrix sizes, since it does not take dependencies into account. This interesting feature of the Cholesky task graph to contain a path with all n POTRF tasks can be used to strengthen the bound, without adding other variables in the linear program. In addition to the n POTRF tasks, this path contains $n-1$ of the $\frac{n\times(n-1)}{2}$ TRSM tasks, and $n-1$ of the $\frac{n\times(n-1)}{2}$ SYRK tasks. We can thus add the following constraint, which states that the execution time is necessarily larger than the time to execute all these tasks in sequence:

$$\sum_r n_{rP} T_{rP} + (n-1) \times T_T^* + (n-1) \times T_S^* \leq l$$

In this constraint, T_{rP} denotes the execution time of POTRF tasks on resource type r, and T_T^* and T_S^* denote the fastest execution time of TRSM and SYRK tasks: we do not model exactly on which resources these TRSM and SYRK tasks are executed, and thus underestimate their completion times, ignoring which resource they actually run on[1]. The resulting lower bound is called the *mixed bound* in the rest of the paper. This linear program has a very small number of variables and constraints (in particular, they are independent of the matrix size), and it can thus be solved very quickly.

B. Constraint Programming formulation

In addition to this lower bound computation, we have used a Constraint Programming formulation of the scheduling problem, in order to obtain good feasible solutions. These solutions provide both a comparison point for StarPU schedules and a limit for possible improvements of the lower bound. The formulation contains one boolean variable b_{ir} for each task i and each resource type r (only one can be true for a given task), and one integer variable s_i for each task i which represents the starting time of the task. The constraints are the following:

$$
\begin{aligned}
& \text{minimize } l \text{ such that} \\
\forall i, \quad & \text{only one type of resource executes task } i\text{:} \\
& \text{OnlyOne}(b_{i1}, \ldots, b_{iR}) \\
\forall i, \quad & \text{task } i \text{ completes:} \\
& s_i + \sum_r b_{ir} T_{ir} \leq l \\
\forall r, \forall t, \quad & \text{at time } \theta \text{ the } M_r \text{ resources of type } r \text{ are} \\
& \text{executing at most } M_r \text{ tasks:} \\
& |\{i \text{ st } s_i \leq \theta < s_i + \sum_r b_{ir} T_{ir}\}| \leq M_r \\
\forall i \to j, \quad & \text{dependency } i \to j \text{ is respected:} \\
& s_i + \sum_r b_{ir} T_{ir} \leq s_j
\end{aligned}
$$

We have implemented this constraint programming formulation using CP Optimizer v12.4. The first constraint is expressed using the `alternative` constraint, and the third constraint uses the concept of *cumulative functions* to express the number of tasks which use resources of type r at time t. The other constraints are simple linear constraints and

[1]It is possible to include additional variables to the linear program to have more precise values, but this does not provide a better bound unless we take more dependencies into account, which requires adding too many variables and constraints and makes the linear program intractable.

are easily expressed. The solver explores the solution space with an exhaustive search and backtracking, using constraint propagation to reduce the search space as much as possible.

Furthermore, providing the result of a HEFT heuristic as an initial solution allows the solver to explore good solutions more rapidly. We let the solver optimize for 23 hours and keep the best solution found in this duration. The obtained solutions are quite good compared to what is obtained with other heuristics, but the solver is unable to prove optimality.

Because it would otherwise be extremely costly to solve [2], this formulation does not take into account data transfers. With the usual platforms and the dense linear algebra operation being studied (the Cholesky factorization), data transfers are indeed not a concern: computation is dense enough for transfers to be largely overlapped with kernel computation.

C. Upper bounds on performance

Lower bounds on execution time also give upper bounds on the performance. Therefore, we have plotted different theoretical performance upper bounds of the Cholesky factorization in Figure 2, based on real execution timings of different tasks on the Mirage machine (described in section V-B).

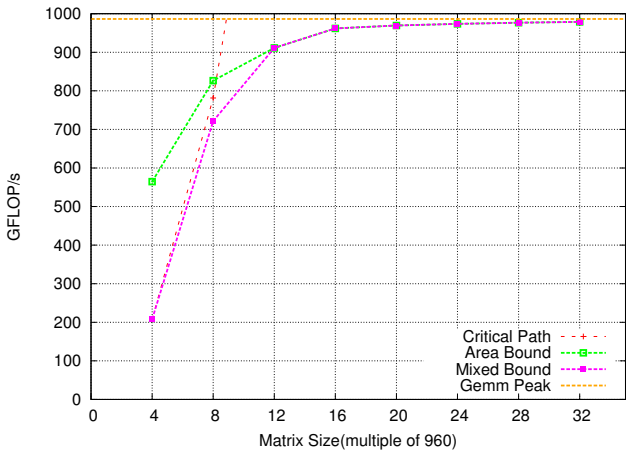


Figure 2. Heterogeneous theoretical performance upper bounds

The critical path bound is calculated based on the critical path of the Cholesky task graph. While calculating the critical path, we have taken into consideration the fastest execution time of each task among the different resources. The *area bound* and *mixed bound* calculations are based on the description given in mixed bound subsection III-A. Since GEMM is the fastest kernel of the Cholesky factorization algorithm, we have also plotted the *GEMM Peak*. This plot shows that the *mixed bound* is the tightest upper bound among all upper bounds, and we will therefore compare the performance of our experiments only with the *mixed bound* in the experiment section.

The performance of the constraint programming solution (best solution found in 23 hours, but not a bound because CP is unable to prove its optimality in 23 hours for matrices larger than 5×5 tiles) described in section III-B will be discussed in section V-C3.

[2]We also have written a version of the constraint programming formulation which takes data transfer times into account but we could not obtain results at the scale of interest for this paper.

IV. Tools and libraries

For this study, we used the Chameleon [19] implementation of the Cholesky factorization, running on top of the StarPU runtime system. We performed real executions on the target platform, and we additionally used the Simgrid [20], [21] simulator, in order to reduce the experimentation time, improve reproducibility of the experiments, and also be able to modify the execution platform.

A. StarPU runtime system

StarPU [1] is a runtime system aiming to allow programmers to exploit the computing power of the available CPUs and GPUs, while relieving them from the need to specifically adapt their programs to the target machine and processing units. The StarPU runtime supports a *task-based programming model*. Applications submit computational tasks, forming a task graph, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. The data that a task manipulates is automatically transferred between the local memory of the accelerators and the main memory, so that application programmers are freed from the scheduling issues and technical details associated with these transfers. In particular, StarPU takes care of scheduling tasks efficiently, using well-known generic dynamic and task graphs scheduling policies from the literature (see Section II-B), and optimizing data transfers using prefetching and overlapping, in particular. In addition, it allows scheduling experts, such as compiler or computational library developers, to implement custom scheduling policies in a portable fashion.

In this study, we specialize the StarPU scheduling algorithms to include a mixture of static and dynamic task assignments, based on the knowledge of the Cholesky task graph, to improve performance on small and medium size matrices. In the following, we call “small” a matrix with less than 10×10 tiles, “medium” a matrix with tile size between 10 and 20, and “large” a matrix with more than 20×20 tiles.

B. Chameleon dense linear algebra library

To cope with the increased degree of parallelism, a new class of dense linear algebra algorithms has been proposed, often referred as tile algorithms in the literature [11], [12].

These algorithms led to the design of new libraries in the past five years such as PLASMA [15], FLAME [22] and DPLASMA [23]. Although both static and dynamic versions of the algorithms have been initially implemented, the dynamic codes are now predominant since they proved to provide more flexibility. These dynamic codes rely on runtime systems (QUARK [2], Supermatrix [4], PaRSEC [3]) that have been specifically designed for the purpose of the numerical software (in the case of PLASMA, FLAME and DPLASMA, respectively).

The advantage of relying on specialized runtime systems is that they can be optimized for both the numerical algorithm and the target architecture. On the other hand, designing and maintaining a runtime system is a highly time consuming task, which makes it difficult to design a fully-featured specialized runtime system. The Chameleon library is based on the PLASMA tiled algorithms and code but relies on the StarPU generic runtime system instead of the specialized QUARK runtime system. One advantage is that it allows for handling heterogeneous architectures (whereas PLASMA and QUARK were initially designed for multicore chips). Another advantage when aiming at focusing on the impact of scheduling strategies is that it allows for running in simulation mode with the field-proven combination [21] of StarPU and Simgrid.

C. Simgrid simulation engine

Simgrid [20] is a versatile simulation toolkit initially designed to study the behavior of large-scale distributed systems like grids, clouds, or peer-to-peer systems. It builds on fluid network models that have been proven as a reasonable alternative to both simple analytic models and expensive, difficult-to-instantiate packet-level simulations.

The Simgrid version of StarPU [21] uses Simgrid to simulate the execution of an application within a single machine. The idea is to run the application normally, except that data transfers and computation kernel calls are replaced by a simple procedure accounting for the time they are expected to take, and gathered coherently by Simgrid. StarPU models each execution unit (CPUs and GPUs) by defining the time taken by each execution unit on each possible task/kernel [24]. It also models the PCI buses between them, using offline bus bandwidth measurements, and relies on Simgrid to compute the interferences on PCI buses between the different transfers.

The resulting simulated times are very close to actual measurements on the real platforms [21], and properly reproduce the various behaviors that can be observed for the various schedulers. This allows one to confidently run experiments with the Simgrid version of StarPU, which provides several advantages:

- The time to simulate execution is reduced, since no actual computation or data transfer is done. The Simgrid simulator itself is not parallel, so the whole execution gets serialized, but several simulations can be run in parallel for e.g. various matrix sizes or schedulers, and one then gets all the results in parallel.
- The experiments do not depend on the availability of the platform, both in terms of quotas, and in terms of versions of the installed software, thus allowing reproducible experiments. This proved useful while performing the experimentation for this very article, since the platform became unavailable for a couple of weeks due to Air Conditioning issues.
- The platform can be modified, for instance to change the available PCI bandwidth, the execution times of the kernels, etc. In Section V-C2, we use this feature in order to build a virtual "related" heterogeneous platform.

In “actual execution mode“, we perform the real execution on Mirage machine (described in section V-B) with StarPU runtime system. While in “simulation mode“, we perform simulation on any machine with Simgrid version of StarPU runtime system by using the configuration files of target platform and expected execution time of kernels on each resource of the target platform.

V. Experiments and Results

A. Schedulers

We have experimented with a few schedulers of StarPU, which are representative of state-of-the-art dynamic heuristics.

The *random* scheduler assigns tasks randomly over all the computation resources. It uses an estimation of the relative performance of the resources as coefficients to balance the randomness, so that GPUs will be assigned more tasks, according to their average acceleration ratio. This is thus representative of classical partitioning heuristics, which take into account the heterogeneity of the platform, but do not take into account the heterogeneity of the tasks.

The *dmda* (deque model data aware) and *dmdas* (deque model data aware sorted) schedulers use the minimum completion time heuristic to assign tasks to computational resources: each task is assigned to the processing resource which is estimated to complete it first, taking into account both the estimated computation time on the estimated target resource, and the possible required data transfer time. The difference between *dmda* and *dmdas* is that *dmdas* schedules tasks in order of their priorities, thus making it representative of the state-of-the-art HEFT heuristic [9], [1].

We are calculating the priorities of different tasks in *dmdas* by estimating the longest path (in terms of execution time) from a task to an exit task (a task without successors) in Cholesky task graph. For longest path calculation, we have taken the fastest execution time of each task among the different resources into consideration.

The Cholesky factorization is a structured application, so we can estimate some extra information in advance by

analyzing the task graph with the help of different tools. This information could be an exact schedule, priorities for some specific tasks, scheduling of some tasks on a particular worker/resource type, etc. In the following section, we inject more or less of these extra information as static knowledge, to influence the scheduling decisions and get better performance.

B. Experimental Setup

We have used a machine called Mirage to run and simulate our experiments. It has 2 Hexa-core Westmere Intel® Xeon® X5650 processors and 3 Nvidia Tesla M2070 GPUs. In the actual execution, we used only 9 CPU cores of the mirage machine so that the remaining 3 CPU cores can be used to fully exploit the critical resource (GPUs) of the system. To make the performance comparable we stick to 9 CPU cores in all of our experiments.

We have used Chameleon v1.0, StarPU v1.2.0 and Simgrid v3.10 for our experiments. We used Intel® Math Kernel Library 11.1, MAGMA 1.4.1 and CUBLAS 6.0 to perform actual executions.

C. Results

We have divided our experiments into two categories based on the types of configuration used. The first one is Homogeneous category where we have run and simulated the performance behavior with 9 homogeneous CPU cores and the second one is Heterogeneous category, where we used 9 CPU cores and 3 GPUs to run the tasks.

From previous work we are getting maximum performance in heterogeneous case with tile size equal to 960 [25], [26], that is why we also kept the same tile size value throughout all our experiments.

For actual executions, we provide the average and standard deviation of 10 runs in the plots. In simulation mode, results are deterministic for all schedulers except for the *random scheduler* which relies on random allocation choices. The simulated plots therefore provide average and standard deviation values of 10 simulations with various seeds for the random scheduler.

1) Homogeneous case: For the homogeneous case, we provide the results of real execution runs of Cholesky factorization with the three different StarPU schedulers: random, dmda and dmdas.

From Figure 3, it is clear that the random scheduler does not perform well. This happens because it does not take into account the already assigned workload of the workers, and just selects a worker among all workers with equal probability. This shows that the scheduler needs to take scheduling decisions in some smart way. The other two schedulers which are based on data aware and early finish time strategies perform much better than the random scheduler. Figure 3 also shows that dmdas slightly underperforms compared to dmda for smaller number of tiles. This

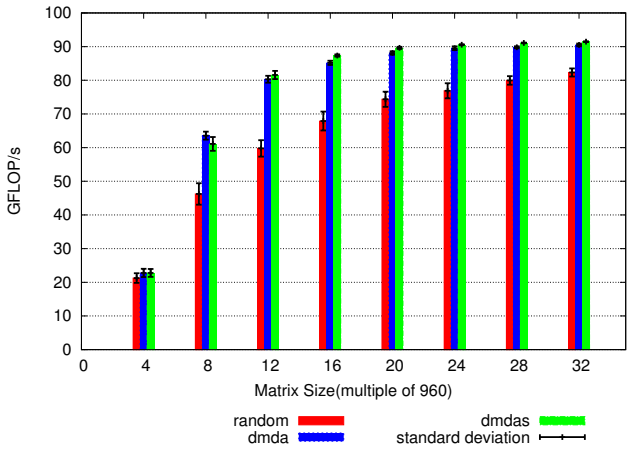


Figure 3. Homogeneous actual performance

is due to the fact that dmdas is biased towards the longest path (path with more work) and chooses some tasks in the beginning which do not generate enough level of parallelism. But as time progresses, *dmdas* starts choosing tasks which releases a higher number of tasks, because these tasks would be the critical ones, which improves the overall performance of the execution.

We are also interested to know what the upper bound of the performance is, in order to determine how far these results are from that bound with different types of schedulers. Since actual executions add some runtime overhead and affect the performance, to mitigate this overhead we have compared the bound with simulated performance.

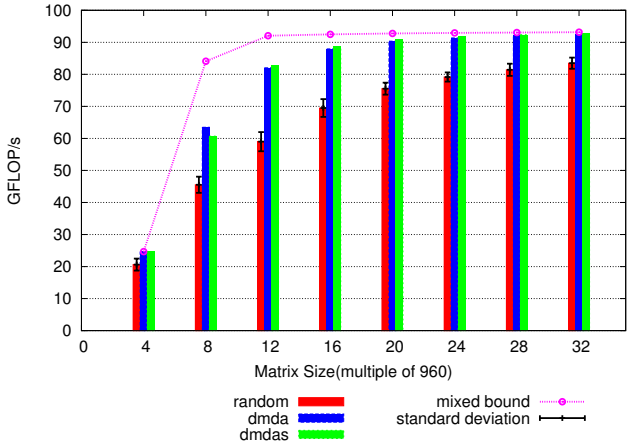


Figure 4. Homogeneous simulated performance

Figure 4 shows that the behavior is very similar to the original execution, with a slight increase in performance, since we have removed the runtime overhead from the simulation. It also shows that the gap between *mixed bound*

and achieved performance is significant for small matrices.

2) Heterogeneous Case: In this subsection, we consider all the processing units of the Mirage machine. 9 CPUs and 3 GPUs are used for the execution of tasks while the remaining 3 CPUs are used as drivers for the 3 GPUs.

Table I
GPUs RELATIVE PERFORMANCE

POTRF	TRSM	SYRK	GEMM
≃2×	≃11×	≃26×	≃29×

Table I shows the GPUs performance for each kernel with respect to CPUs performance, e.g.: GEMM is 29 times faster on GPU compared to CPU.

We divide our work into two parts. In the first part, we consider the impact of heterogeneity of resources by considering a heterogeneous platform with related performance. More specifically, we designed a fictitious hardware configuration, where execution time of each kernel on GPU is made to be exactly K times faster than the CPU execution time, and we call this case the *heterogeneous related*. The common accelerator factor K is an average over the actual measured acceleration factors, computed as follows :

$$K = \left(\frac{N_P * a_P + N_T * a_T + N_S * a_S + N_G * a_G}{\text{Total Number of Tasks}} \right)$$

where,
N_P: total number of POTRF tasks
a_P: acceleration factor of POTRF on GPU
N_T: total number of TRSM tasks
a_T: acceleration factor of TRSM on GPU
N_S: total number of SYRK tasks
a_S: acceleration factor of SYRK on GPU
N_G: total number of GEMM tasks
a_G: acceleration factor of GEMM on GPU

Here, the acceleration factor depends on the number of tasks and the number of tasks depends on the number of tiles. Therefore, we get different acceleration factors with different number of tiles. Acceleration factors for 4, 8, 12, 16, 20, 24, 28 and 32 tiles matrices are 17.30, 22.30, 24.30, 25.38, 26.06, 26.52, 26.86 and 27.11 respectively.

In the second part of our work, we show the achieved performance with the actual hardware with the help of both actual and simulated executions, and we call this case the *heterogeneous unrelated* case.

We are using the *mixed bound* (as explained in Section III-A) to compare the performance. The bounds do not take into account the communication constraints. Therefore, to be fair in the comparison we have used the simulated performance, where communication costs have been removed by modifying the platform file of our machine (one of the interesting features of the Simgrid version of the StarPU runtime system).

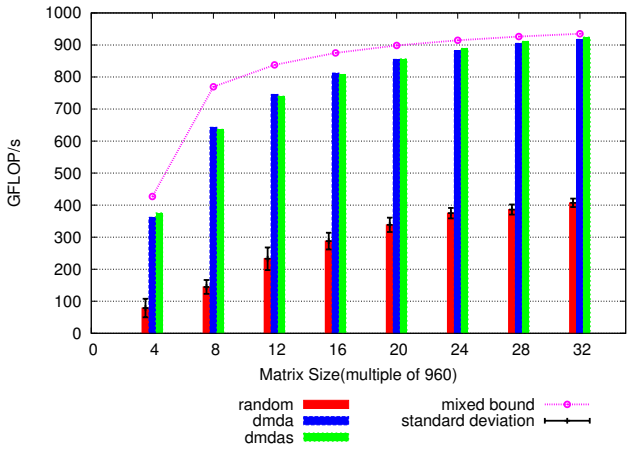


Figure 5. Heterogeneous related simulated performance

Heterogeneous related case: Figure 5 shows the simulated performance with different schedulers on the fictitious heterogeneous platform. Here, we can see that the random scheduler performs very poorly because it assigns tasks randomly to the worker without knowing the already assigned workload of workers, which limits the number of ready tasks in the system, and introduces significant idle time on our critical resource (GPUs). We have also computed the *mixed bound* for this fictitious platform. The difference between simulated performance and *mixed bound* is once again significant for small and medium size matrices.

Heterogeneous unrelated case: First we compare the performance of different schedulers in actual execution and then between simulated performance and mixed bound.

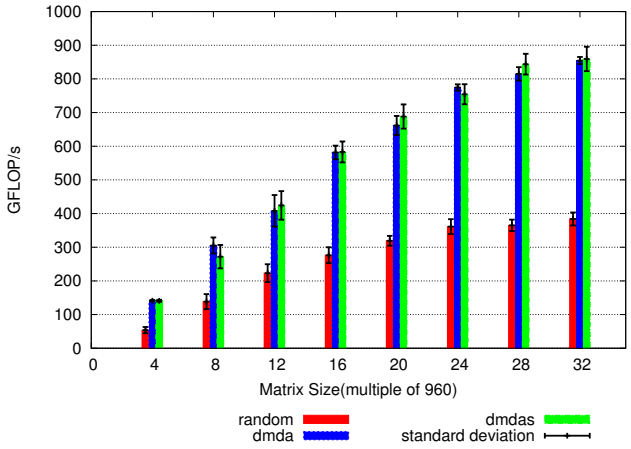


Figure 6. Heterogeneous unrelated actual performance

As shown in Figure 6, in actual executions, the *random* scheduler does not perform well because it is not taking data movement into account while making scheduling decisions:

it assigns worker randomly for each task, which may select different resource types for data dependent tasks and result in lots of data movement from CPU memory to GPU memory and vice-versa. In addition, it is also not taking the affinity of tasks to resource (e.g.: GEMM/SYRK is more suitable to be executed on GPU) into account, which degrades the overall performance of the system. The other two schedulers perform comparatively better than the random scheduler because they take into account data transfers when assessing completion time in the HEFT-like scheduling strategies. Here we can also see that *dmda* outperforms *dmdas* performance for some matrices, for the same reason as for the homogeneous case (choosing the critical task versus tasks which generate high level of parallelism).

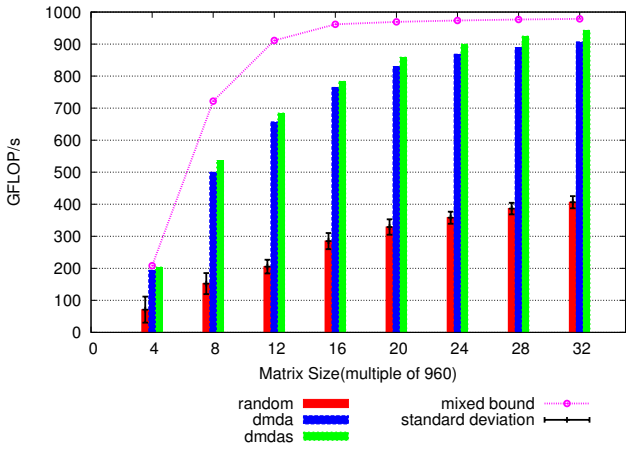


Figure 7. Heterogeneous unrelated simulated performance

We are now again interested in determining how far we are from the peak performance of the application. Thus, we performed the simulation with different numbers of tiles. Figure 7 illustrates the comparison between bounds and achieved performance in simulation. Here we can also see that the performance difference between the best scheduler and the mixed bound is significant for small and medium size matrices.

Comparison between Heterogeneous related and unrelated case: In order to determine the impact of heterogeneity of speed-up of tasks on performance, we present a comparison between related and unrelated heterogeneous simulations. To this end, we scaled the mixed bound of the related case such that it perfectly matches with the mixed bound of the unrelated case, and also scaled all the performance values of the related case with the same factor. The obtained results are given in Figure 8, which can now be compared with the unrelated case of Figure 7.

Here we can see that unrelated speed-ups make the problem harder. That is why the gap between state-of-the-art schedulers performance and *mixed bound* is large in Figure 7 compared to Figure 8. Here, it is also clear that there is

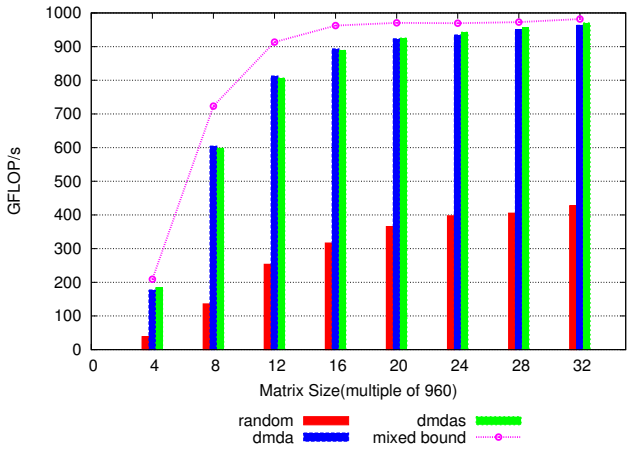


Figure 8. Heterogeneous related simulated *scaled performance*

room for improvement in the case of small and medium size matrices in the heterogeneous case.

3) Scheduling with static knowledge: The significant gap between the performance of StarPU schedulers and the theoretical bound (*mixed bound*) for small and medium size matrices in Figure 7 highlights the following things:

- Either the dynamic schedulers of StarPU return a scheduling that can be improved for small and medium size matrices;
- Or the theoretical bound is not tight enough;
- Or both.

Indeed, the *dmda* and *dmdas* schedulers take only dynamic decisions to map the ready tasks onto the processors depending on the state of resources and estimation of execution and communication times (also priorities among ready tasks in *dmdas*), without taking into account the overall task graph. These local choices may lead to bad decisions when the parallelism in the task graph is limited. We conducted some experiments to improve the overall performance with static information in the heterogeneous unrelated case.

Since GEMM and SYRK kernels are well suited to execute on GPUs, we enforced these kernels to be executed on GPUs as static information to the StarPU runtime system. This strategy improves the performance slightly for some matrices in simulation but the performance improvement was not significant and the reason for this is that the StarPU schedulers (*dmda* and *dmdas*) already choose GPUs to execute most of the GEMM and SYRK kernels.

We also analyzed the solution of the *mixed bound* and noticed that a significant portion of the TRSM kernels were mapped onto CPUs. Analyzing traces generated by *dmda* and *dmdas* schedulers reveals that both policies allocate very few TRSMs on CPUs. Since the *mixed bound* does not take all dependencies into account, it is not clear which TRSM kernels should be executed on CPUs in order to

improve the performance. On the Mirage machine, with real timings of tasks, we found that the critical path of the Cholesky factorization passes through the diagonal and second diagonal tiles (sequence of *POTRF* $\rightarrow$ *TRSM* $\rightarrow$ *SYRK* $\rightarrow$ *POTRF* ..... $\rightarrow$ *SYRK* $\rightarrow$ *POTRF*). Therefore, we have evaluated the performance in simulation with *dmdas* scheduler where all the TRSM kernels which are at least k ($1 \leq k <$ *Number of Tiles*) tiles away from the diagonal are forced to execute on the CPUs (see Figure 9) and plotted the best obtained performance in Figure 10. We obtained best performance when all the TRSM kernels which are more than 6-8 tiles away from the diagonal are forced on CPUs.

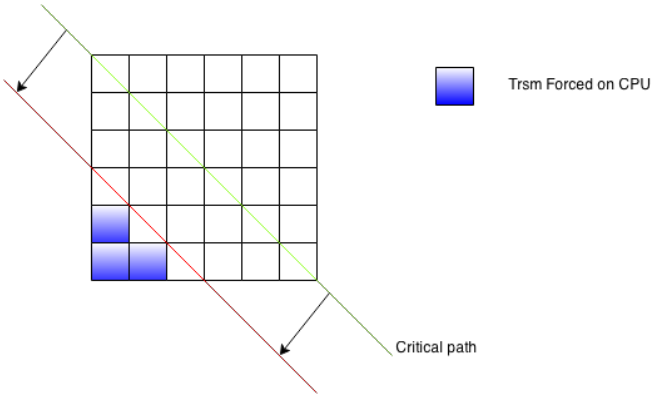


Figure 9. TRSMs forced on CPUs

Figure 10 shows that providing information about the TRSMs triangular structure statically allows one to achieve better performance than present state-of-the-art schedulers for small and medium size matrices.

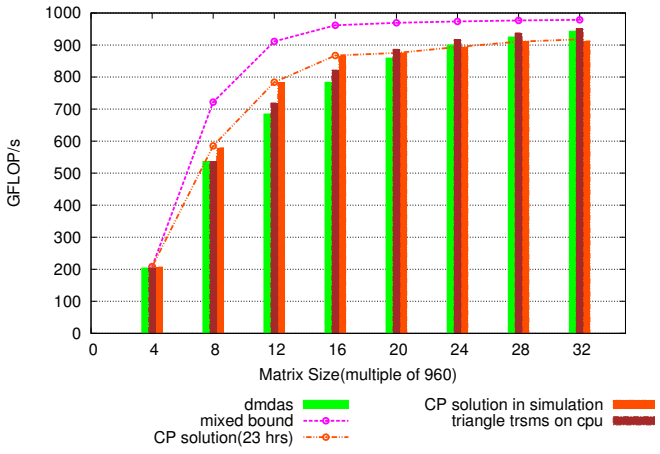


Figure 10. Heterogeneous unrelated simulated performance with static knowledge

We eventually used the constraint programming (CP) described in Section III-B to find an optimal solution and ran it for 23 hours, but unfortunately we did not manage to get an optimal solution, particularly for large matrix sizes, which produce a very large constraint program (and thus this is not a performance bound). Nevertheless, for reasonable matrix sizes, it provides good and feasible solutions in that span of time. Theoretical performance value with CP solution (*CP solution(23 hrs)* in Figure 10) was better than the values what we are getting with state-of-the-art schedulers in simulation for small and medium size matrices. We thus injected the exact schedule obtained from CP solution in the simulation and obtained almost equal (difference is less than 1%) performance (*CP solution in simulation* in Figure 10) compared to theoretical performance value, which also shows the robustness of the Simgrid version of StarPU with simulation.

Performance improvement obtained in simulation by injecting static information to scheduler motivated us to conduct some actual execution with static information. Therefore, we conducted some actual execution by injecting the triangular information, force all TRSMs on CPUs which are at least k ($1 \leq k <$ *Number of Tiles*) tiles away from diagonal as static knowledge. Figure 11 shows the best obtained performance in actual execution among performance obtained with all possible values of k.

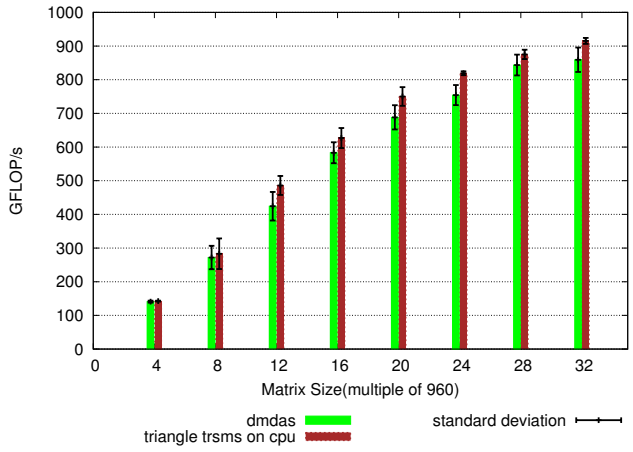


Figure 11. Heterogeneous actual performance with static knowledge

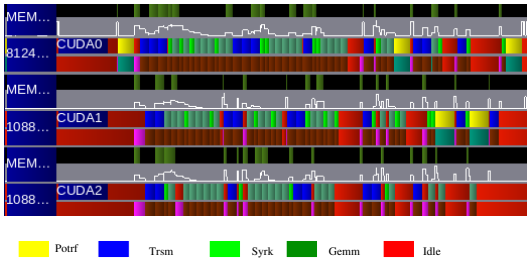
We also conducted some experiments by injecting the CP schedule in actual execution for small matrices, however we did not achieve good performance improvement compared to what we are achieving in simulation. The CP formulation indeed does not account data transfers, since as described in Section III-B, solving a CP with data transfers has shown intractable for the purpose at stake. Actual execution with CP schedule thus adds lots of idle time on resources during data transfer, and consequently does not reproduce the same performance in actual execution. The simulated execution has however allowed us to show, at least in the case without data transfers, that some heuristics get relatively close to an

achievable CP solution. We are currently extending the CP formulation to partially take data transfers into account, so that it can be used for real executions, but this is beyond the scope of this paper.

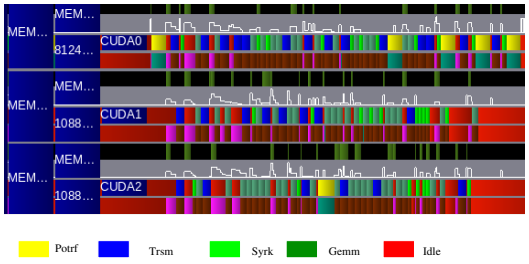
VI. Discussion

A. *dmda vs dmdas scheduler*

We were expecting that *dmdas* would always perform better than *dmda* scheduler because it is also taking the HEFT priorities into account while making scheduling decision. Nevertheless, we found a few cases where *dmda* outperforms *dmdas*. We investigated the generated trace files with *dmda* and *dmdas* schedulers in order to determine the reasons of this behavior and we found that *dmdas* puts emphasis on critical path rather than parallelism, since it selects some tasks in the beginning which are critical but are not generating enough level of parallelism. That introduces some idle time on the critical resource (GPUs) and degrades the overall performance of the system, which is a known defect of the HEFT scheduler in general. Figure 12 shows traces with *dmda* and *dmdas* schedulers.



(a) GPU trace with dmda scheduler



(b) GPU trace with dmdas scheduler

Figure 12. GPU Traces for 8 × 8 tiles

B. *Mapping from Constraint Programming solution*

We conducted some experiments in simulation by injecting only the mapping information (i.e. only the CPU/GPU information, not the exact task order) of the feasible solution statically obtained by constraint programming, and let the scheduler decide the precise ordering and worker dynamically. This extra information about resource allocation did not improve the performance of the system compared to the performance obtained by *dmda* and *dmdas* schedulers, which indicates that the feasible solution is highly dependent of the precise ordering chosen by constraint programming. This shows that heuristics required to achieve this performance are probably very complex, probably even beyond only backfilling.

VII. Conclusion

In this work, we have bridged the gap between theoretical performance bounds and actually achieved performance on the dense Cholesky factorization. On the former side, we have proposed improved bounds which take into account both resource and task heterogeneity, as well as critical paths. On the latter side, we have introduced some static information into the dynamic task scheduler of StarPU, which brought the performance closer to the theoretical bounds, and very close to what a statically-optimized schedule can achieve. We have also shown that the performance achieved by such statically-optimized schedule depends on precise non-intuitive task ordering, which thus can not be reached by simple list-scheduling heuristics, even with backfilling.

We will verify the results on other hardware platforms, and apply the same methodology to other dense linear algebra algorithms, but we also plan to try other classes of applications, notably more irregular applications such as sparse linear algebra or FMM (Fast Multipole Method).

More generally, this work opens a bridge to closer interaction between applications and tasks schedulers. We have shown that while generic heuristics such as HEFT achieve very good performance, application-specific scheduling hints can noticeably improve performance. We aim at generalizing and formalizing this type of information, so that scheduling experts can easily analyze achieved performance, optimize the schedule statically, and try to inject more or less application-specific scheduling hints into the scheduler, such as "this proportion of TRSM tasks should be run on CPUs", or "these TRSM tasks should be run on CPUs", etc. The code produced for the purpose of the study will be reversed in the StarPU runtime system and Chameleon dense linear algebra library.

Acknowledgment

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, being developed under the INRIA PlaFRIM development action with support from LaBRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS.

This work has been partially supported by the *ANR SOLHAR* project, funded by the French Research Agency.

References

[1] 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.

[2] A. YarKhan, J. Kurzak, and J. Dongarra, *QUARK Users' Guide: QUeueing And Runtime for Kernels*, UTK ICL, 2011.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[4] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. Van de Geijn, "SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, p. 123–132.

[5] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[6] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *2007 International Workshop on Parallel Symbolic Computation*, ser. PASCO '07. ACM, 2007, pp. 15–23.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[8] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

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

[10] F. Gustavson, "High-performance linear algebra algorithms using new generalized data structures for matrices," *IBM Journal of Research and Development*, vol. 47, no. 1, pp. 31–55, Jan 2003.

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

[12] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Transactions on Mathematical Software (TOMS)*, vol. 36, no. 3, p. 14, 2009.

[13] C. D. Dhillon, J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "Lapack working note 95 scalapack: A portable linear algebra library for distributed memory computers - design issues and performance," 1995.

[14] C. H. Koelbel, *The High performance Fortran handbook*, ser. Scientific and engineering computation. Cambridge, Mass. MIT Press, 1994.

[15] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Lapack working note 191: A class of parallel tiled linear algebra algorithms for multicore architectures," 2007.

[16] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, 1990.

[17] G. Manimaran and C. S. R. Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 3, pp. 312–319, 1998.

[18] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessing*, ser. Research monographs in parallel and distributed computing. London: Pitman, 1989.

[19] "Chameleon, a dense linear algebra software for heterogeneous architectures," 2014. [Online]. Available: https://project.inria.fr/chameleon

[20] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *10th IEEE International Conference on Computer Modeling and Simulation (UKSim)*, Apr. 2008.

[21] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures," in *Euro-par - 20th International Conference on Parallel Processing*, Aug. 2014.

[22] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. V. Zee, "The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations," *J. Parallel Distrib. Comput.*, vol. 72, no. 9, pp. 1134–1143, 2012.

[23] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA," ICL, UTK, Tech. Rep., 2010.

[24] C. Augonnet, S. Thibault, and R. Namyst, "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures," in *International Euro-Par Workshops 2009, HPPC'09*, ser. Lecture Notes in Computer Science, vol. 6043. Delft, The Netherlands: Springer, Aug. 2009, pp. 56–65.

[25] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in *GPU Computing Gems*, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2.

[26] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR factorization on a multicore node enhanced with multiple GPU accelerators," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011*, 2011, pp. 932–943.