Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources

Suraj Kumar

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Suraj Kumar**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources

---

Date de soutenance : 12 April 2017

Devant la commission d'examen composée de :

Mme Padma RAGHAVAN	Professeur, Vanderbilt University	Rapporteur
M. Julien LANGOU	Professeur, University of Colorado Denver	Rapporteur
M. Denis TRYSTRAM	Professeur, Grenoble Institute of Technology	Président
Mme Veronika SONIGO	Maître de conférences, FEMTO-ST institute	Examinateur
M. Olivier BEAUMONT	DR1, Inria Bordeaux	Directeur de Thèse
M. Lionel EYRAUD-DUBOIS	CR, Inria Bordeaux	Directeur de Thèse
M. Samuel THIBAULT	MCF, University of Bordeaux	Directeur de Thèse
M. Emmanuel AGULLO	CR1, Inria Bordeaux	Directeur de Thèse
M. Abdou GERMOUCHE	MCF, University of Bordeaux	Invité Membre

– 2017 –

This page is intentionally left blank.

Résumé Du fait des énormes capacités de calculs des accélérateurs tels que les GPUs et les Xeon Phi, l'utilisation de machines multicoques pourvues d'accélérateurs est devenue commune dans le domaine du calcul haute performance (HPC). La complexité induite par ces accélérateurs a suscité le développement de systèmes d'exécution à base de tâches, dans lesquels les dépendances entre les applications sont exprimées sous la forme de graphe de tâches et où les tâches sont ordonnancées dynamiquement sur les ressources de calcul. La difficulté est alors de concevoir des stratégies d'ordonnancement qui font une utilisation efficace des ressources de calculs et le développement de telles stratégies, même pour un unique noeud hybride, est un enjeu essentiel de la performance des systèmes HPC.

Nous considérons dans cette thèse l'ordonnancement de noyaux d'algèbre linéaire dense sur des noeuds complètement hétérogènes et constitués de CPUs et de GPUs. Les performances relatives des accélérateurs par rapport aux coeurs classique dépend très fortement du noyau considéré. Par exemple, les accélérateurs sont beaucoup plus efficaces pour les produits de matrices, par exemple, que pour les factorisations. Dans cette thèse, nous analysons les performances de stratégies statiques et dynamiques d'ordonnancement et nous proposons un ensemble de stratégies intermédiaires, en ajoutant des composantes statiques (respectivement dynamiques) à des stratégies d'ordonnancements dynamique (respectivement statiques). Récemment, une stratégie appelée HeteroPrio a été proposée, qui s'appuie sur les affinités entre les tâches et les ressources pour un petit ensemble de tâches différentes s'exécutant sur deux types de ressources. Nous avons étendu cette stratégie d'ordonnancement pour des graphes de tâches généraux pour deux types de ressources puis pour plus de deux types. De manière complémentaire, nous avons également démontré des facteurs d'approximation et des pires cas pour HeteroPrio dans le cas d'un ensemble de tâches indépendantes sur différents types de plates-formes.

Mots-clés Algèbre linéaire dense, systèmes d'ordonnancement dynamiques, plates-formes hétérogènes, ordonnancement à base de graphe de tâches, ordonnancement dynamique

Laboratoire d'accueil Laboratoire Bordelais de Recherche en Informatique, Bordeaux, France

This page is intentionally left blank.

Title Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources

Abstract Due to massive computation power of accelerators such as GPU, Xeon phi, multicore machines equipped with accelerators are becoming popular in High Performance Computing (HPC). The added complexity led to the development of different task-based runtime systems, which allow computations to be expressed as graphs of tasks and rely on runtime systems to schedule those tasks among all resources of the platform. The real challenge is to design efficient schedulers for such runtimes to make effective utilization of all resources. Developing good schedulers, even for a single hybrid node, and analyzing them can thus have a strong impact on the performance of current HPC systems.

We consider the problem of scheduling dense linear algebra applications on fully hybrid platforms made of CPUs and GPUs. The relative performance of CPU and GPU highly depends on the sub-routine. For instance, GPUs are much more efficient to process matrix-matrix multiplications than matrix factorizations. In this thesis, we analyze the performance of static and dynamic scheduling strategies and we propose a set of intermediate strategies, by adding static (resp. dynamic) features into dynamic (resp. static) strategies. A resource centric dynamic scheduler, HeteroPrio, which is based on affinity between tasks and resources, has been proposed recently for a set of small independent tasks on two types of resources. We extend and analyze this scheduler for general task graphs first on two types of resources and then on more than two types of resources. Additionally, we provide approximation ratios and worst case examples of HeteroPrio for a set of independent tasks on different platform sizes.

Keywords Dense Linear Algebra; Runtime Systems; Heterogeneous Platforms; Task-based Scheduling; Dynamic Schedulers;

This page is intentionally left blank.

Acknowledgements

I would like to express my sincere gratitude to Olivier Beaumont for his excellent supervision and constant support throughout the course of this thesis. I feel extremely privileged to get so much of his time and to work with him throughout these years. I also thank my co-advisors Lionel Eyraud-Dubois, Samuel Thibault and Emmanuel Agullo for their fruitful collaborations and very interesting discussions. I also wish to thank Abdou Guermouche for his support during this thesis.

I would like to thank all the members of the thesis committee, especially reporters Padma Raghavan and Julien Langou for reviewing my dissertation and providing me very useful remarks, questions and suggestions for future directions of this work. I also thank examiners Denis Trystram and Veronika Sonigo for providing detailed feedback on the dissertation.

I convey my sincere thanks to the team leader Denis Barthou and all the researchers of the team for emphasizing the importance of research and creating a wonderful research environment in STORM team, Inria, Bordeaux.

I am grateful to my family members for their constant motivation, love and support. I would like to thank my colleagues and former colleagues especially Stojce Nakov, Marc Sergent, Luka Stanisic, Florent Pruvost and Terry Cojean for their support during my course at Inria. I would also like to thank all my friends for making my stay at Inria memorable and accompanying me in difficult times.

I express my gratitude to Inria, its founders and the people who helped shape Inria over the years, for providing me a platform to pursue great endeavors in my life.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Introduction

The increasing need to process large amount of computations and to analyze large data in real life with quality and accuracy encouraged the use of High Performance Computing (HPC) systems significantly in recent years. HPC refers to the use of aggregated computing power in a way which delivers much higher performance than one can get on a typical desktop for large problems. HPC provides the ability to analyze and to understand the complexity of a huge amount of information, coming from different sources, and helps us to solve some challenges of our society. In recent years, Cloud computing is becoming increasingly popular in HPC area. It provides a cost effective model of utilization of computing infrastructures. Compute resources, storage resources, even applications can be procured on pay-per-use basis.

HPC is widely used to provide public safety in emergency scenarios. For instance, it allows us to predict the size and patch of storms and flood more precisely and further in advance, which helps us to take preventing measures and to reduce damage. HPC also provides substantial benefits in healthcare [89, 55]. It allows us to design and simulate the effect of new drugs, to provide faster diagnosis and better treatment. It helps to detect genetic changes responsible for the onset and mutation of tumors in a simple, quick and precise way. Consider for instance the new born babies with genetic disorders – the main cause of infant death, time is essential as they do not clearly show all of the classical symptoms that make diagnosis possible. HPC allows us to analyze a large set of nucleotides (building blocks of nucleic acids) sequences in a few hours and enables us to provide effective treatment.

HPC is also used in finance market to manage assets and risks. Several companies use supercomputers to measure risks in their fixed income operations by assessing tens of thousand of possible market scenarios [55]. Entertainment field also relies heavily on HPC in order to make animated movies. For instance, to make the movie **Avatar** [3], 40,000 processors were handling around 8 gigabytes of data per second, running 24 hours a day [76].

In last few decades, we collected a large amount of data, which increases the need to analyze the data. HPC can be an useful tool in this area as well. Researchers from different fields such as social media, geology, archeology, materials, graphics, genomics, brain imaging, economics, oil and gas, space, nuclear, even music use HPC platforms to conduct their research [101].

Most applications running on supercomputers such as weather prediction, seismic imaging, nuclear simulation use different linear algebra subroutines. Therefore, improving performance of these linear algebra subroutines has become important since 1970. A specification for these linear algebra subroutines using scalars and vectors, Basic Linear Algebra Subroutines (BLAS) level-1 was published in 1979. To take advantage of vector processors, BLAS was augmented with level 2 operations that perform matrix-vector operations. To take advantage of cached memory, in 1987, level 3 BLAS operations were introduced that perform matrix-matrix computations. Many linear algebra libraries use BLAS libraries to perform linear algebra computations. LINPACK [1] library developed in late 1970s uses BLAS level 1 subroutines. LAPACK [19], released in 1992, is the successor of LINPACK and uses BLAS level 3 operations to exploit caches of modern architectures. LINPACK Benchmark [56] which is initially designed to estimate the performance of a system using LINPACK library, is still used to measure the performance of modern supercomputers. The benchmark used in LINPACK Benchmark is to solve a dense system of linear equations with LU factorization using partial pivoting. The TOP500 list ranks the supercomputers twice a year since June 1993 based on their performance on the LINPACK Benchmark [11].

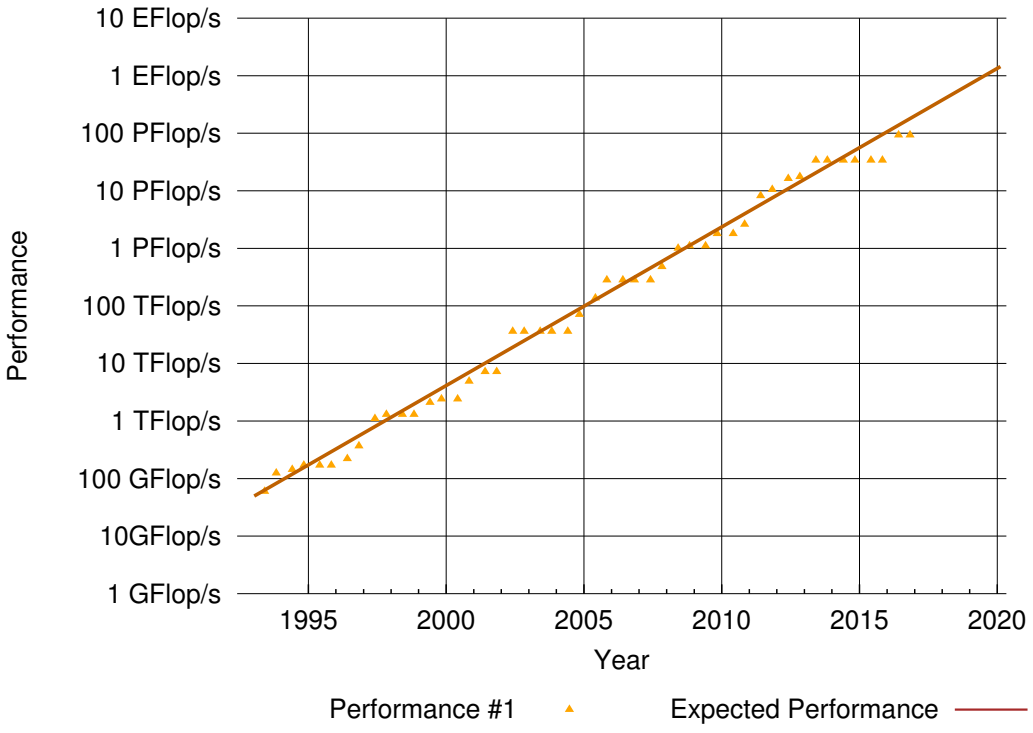


Figure 1: Expected performance of the fastest supercomputer.

Presently the fastest supercomputer in the world is Sunway TaihuLight,

developed at National Supercomputing Center in Wuxi, China. It achieve 93 PFlop/s performance on LINPACK Benchmark, while its theoretical peak is 125 PFlop/s [10]. Figure 1 shows the projected performance of fastest supercomputer for next few years [9]. It exhibits that the fastest supercomputer is expected to achieve exascale performance by 2020. Many countries, such as China, US, Japan, France have plans to deploy their exascale supercomputers in next few years. China is scheduled to deploy a prototype of an exacale computer this year and expected to field it in 2020 [78]. Japan has also plans to bring its first exascale computer by 2020/2021. France is also scheduled to get its first exacale machine deployed at CEA, the French Atomic Energy Agency, by 2020. Paul Messina, head of the US Department of Energy's Exascale Computing Project, recently announced that US will deploy initial exascale system sometime in 2021 [77].

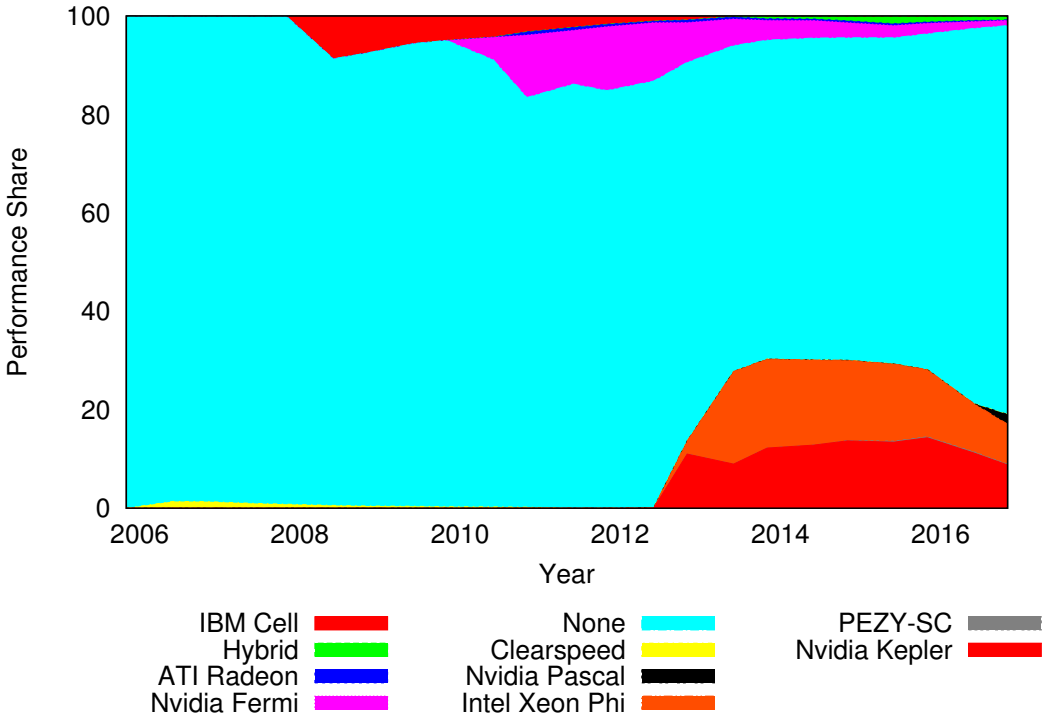


Figure 2: Performance share of different accelerators in TOP500 list.

Massive computation power of accelerator makes them to consider for supercomputers. Figure 2 shows the performance share of accelerators in the TOP500 list. The idea of using accelerators is not new (FPGAs were used in past as coprocessors), it is refurbished in recent years due to huge processing capabilities of the different accelerators. For example, Roadrunner supercomputer built by IBM, which had AMD Opteron processors and cell accelerators, ranked #1 in TOP500 list of June 2008 [2]. The use of accelerators for HPC

community however gained popularity with the advent of GPUs. Initially GPUs were used in rendering, which involves a large amount of computations. This led researchers to think that these devices could be used to accelerate scientific computing applications as well, especially where computation dominates latency, such as for dense linear algebra.

Figure 2 exhibits that presently 21 % performance share of TOP500 list is produced by accelerators. It also indicates that significant amount of computers in TOP500 list is based on hybrid architecture. Most of these systems are using Nvidia GPUs or Intel Xeon Phi coprocessors. The faster US supercomputer, Titan (3rd rank in TOP500 list) is also a hybrid supercomputer, which has AMD Opteron CPUs and Nvidia GPUs [7]. Optimizing performance of a complex computation on such hybrid architectures is very complex. Developing good scheduling algorithms, even on a single hybrid node, and analyzing them can thus have a very high impact on the performance of current HPC systems. This is the goal of this thesis.

Accelerators such as GPUs are employed in processing nodes usually beside multicores. When trying to exploit both CPUs and GPUs, users face several issues. Indeed, several phenomena are added to the inherent complexity of the underlying NP-hard optimization problem.

First, multicores and GPUs are unrelated resources, in the sense that depending on the targeted computation, the performance of the GPUs may be much higher, close or even worse than the performance of a CPU. In the literature, unrelated resources are known to make scheduling problems harder (see [41] for a survey on the complexity of scheduling problems, [73] for the specific simpler case of independent tasks scheduling and [31] for a recent survey in the case of CPU and GPU nodes). Second, the number of available architectures has increased dramatically with the combination of available resources (both in terms of multicores and accelerators). Therefore, it is almost impossible to develop optimized hand tuned kernels for all these architectures. Third, nodes have many shared resources (caches, buses) and exhibit complex memory access patterns (NUMA effects), that render the precise estimation of the duration of tasks and data transfers extremely difficult.

All these characteristics make it hard to design scheduling and resource allocation policies even on very regular kernels such as linear algebra. On the other hand, this situation favors dynamic strategies where decisions are made at runtime based on the state of the machine and on the knowledge of the application (to favor tasks that are close to the critical path for instance). In recent years, several task-based systems have been developed such as StarPU [22], StarSs [82], SuperMatrix [46], QUARK [104], XKaapi [67] or PaRSEC [36]. All these runtime systems model the application as a Direct Acyclic Graph (DAG), where vertices correspond to tasks and edges to dependencies between these tasks. Figure 3 shows an example of a DAG, where vertices a, b, c, d, e and f represent tasks, and edges ac, ad, bc, bd, be, df and ef represent depen-

dencies. At runtime, the scheduler knows (i) the state of the different resources (ii) the set of tasks that are currently processed by all non idle resources (iii) the set of (independent) tasks whose all dependencies have been solved (iv) the location of all input data of all tasks (v) possibly an estimation of the duration of each task on each resource and of each communication between each pair of resources and (vi) possibly priorities associated to tasks and that have been computed offline. Based on this information, scheduler takes scheduling and allocation decisions. HEFT heuristic [99] is certainly the most popular of this class of algorithms.

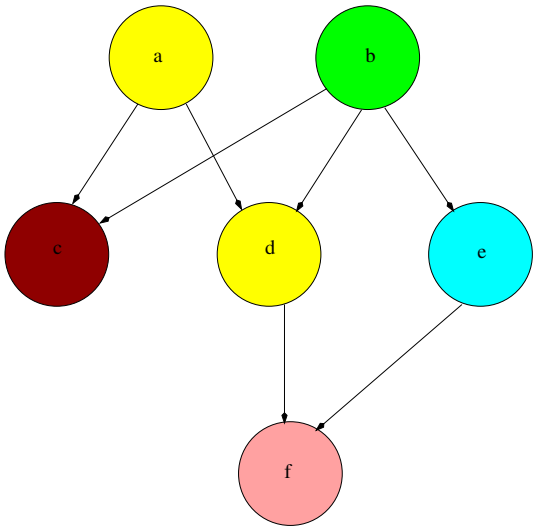


Figure 3: A DAG.

In this thesis, we consider the scheduling problems of task based dense linear algebra kernels on a single hybrid node composed of CPUs and GPUs. The contributions of this thesis therefore cover the different aspects of scheduling which must be addressed at the level of the runtime system. More particularly we identify the following contributions.

- **Performance comparison between static strategies with dynamic corrections and dynamic strategies with static information.** We provide a fair performance comparison between static and dynamic strategies and we propose a set of intermediate strategies by adding more static (resp. dynamic) features into dynamic (resp. static) strategies. We also exhibit that adding simple static information about applications improves performance of dynamic schedulers significantly.

- **Theoretical performance upper bounds of task graphs.** It is well known that system peak is hard to achieve. Performance of any task based application is limited by its task types and dependencies among tasks. We provide some theoretical performance bounds by considering

heterogenity of tasks and resources as well as some dependencies. Performance bounds on task graphs help us to assess the quality of different schedulers.

- **Resource centric schedulers for task graphs.** A resource centric scheduler, *HeteroPrio* was proposed recently for a set of small independent tasks on two types of resources, which is based on affinity between tasks and resources [16]. We extend this scheduler for general task graphs. On two types of resources, affinity can be expressed as task acceleration factor but acceleration factor does not make sense when we have more than two types of resources. We consider different heuristics to define affinity and then generalize HeteroPrio to multiresource case. This scheduler is very effective for applications where scheduling decisions are very important, such as Cholesky factorization of medium size matrices.

- **Approximation ratios of HeteroPrio.** We provide approximation bounds of HeteroPrio compared to the optimal schedule in the case where all tasks are independent and for different platform size. We also provide worst case examples for different platform sizes and prove that almost all our bounds are tight.

The outline of the thesis is the following. Chapter 1 describes different state-of-the-art task based runtime systems, linear algebra libraries and simulators. In Chapters 2 and 3, we analyze different static and dynamic strategies and provide different performance bounds of task graphs. We extend HeteroPrio to multiple resources in Chapter 4. In Chapter 5, we provide approximation ratios of HeteroPrio on two types of resources.

The main contributions of different chapters are presented in the following paragraphs.

In Chapter 2, we concentrate on the analysis of the behavior of Cholesky factorization on a heterogeneous node consists of CPUs and GPUs. We show how adding simple static rules based on an offline analysis of the problem (such as processing of tasks which are far from critical path on slow resources) into dynamic schedulers improves the overall performance of the application. We also provide theoretical bounds on the performance of task graphs for a given platform. This work has been conducted in collaboration with Julien Herrmann and Loris Marchal from ENS Lyon, France.

In Chapter 3 we propose different scheduling strategies by adding more static (resp. dynamic) features into dynamic (resp. static) strategies. We propose a dynamic strategy, HeteroPrio, which is based on the acceleration ratio on GPU to establish affinity between the resources and the different types of tasks, for general task graphs. In order to fully exploit the heterogeneous resources, GPUs should preferably execute tasks with higher acceleration factors,

and CPUs should execute tasks with lower acceleration factors. HeteroPrio must be associated to a spoliation mechanism. Indeed, in above description, nothing prevents the slow resource to execute a task for which it can be arbitrarily badly suited, thus leading to arbitrarily bad results. Therefore, when a fast resource is idle and would be able to restart a task already started on a slow resource and to finish it earlier than on the slow resource, then the task is spoliated and restarted (not preempted) on the fast resource. We propose several corrections to HeteroPrio, such as fast (resp. slow) workers select the highest (resp. lowest) priority ready tasks and tasks whose acceleration factors are in a relatively thin range of values are treated equally, to find the best trade-off between acceleration of tasks and progress.

In Chapter 4, we extend HeteroPrio algorithm for more than two types of resources. Since HeteroPrio is based on the notion of heterogeneity, we proposed two heuristics to determine the heterogeneity score of a task on a resource. First heuristic is based on the area solution of the task graph for a given platform. It provides a generic way of detecting which tasks are more suited to which resources. Second heuristic is based on idea of how "good" this resource is compared to the worst one, and how "bad" it is compared to the best one. We exhibited that these heuristics are efficient even in highly heterogeneous configurations and outperform HEFT-based strategy significantly. This work has been conducted in collaboration with Terry Cojean, another PhD student of my research team, and Abdou Guermouche from HiePACS team, Inria Bordeaux, France.

In Chapter 5, we provide approximation ratios and worst case examples for HeteroPrio in the case where all tasks are independent. Interestingly, we show that spoliation allows to prove approximation ratios for a list scheduling algorithm on two unrelated resources, which is not possible otherwise. We also establish that almost all our approximation ratios are tight.

This page is intentionally left blank.

Chapter 1

Background

In this chapter, we describe different tasks based runtime systems which allow programmers to express applications at high level with simple APIs and relieve them from the burden of dealing with low-level details such as prefetching, data transfers, scheduling of tasks, or synchronizations. Runtime systems employ a very modular approach. Applications are expressed as directed acyclic graphs (DAG) of tasks, where vertices represent tasks to be executed and edges represent dependencies between those tasks. We also describe a framework to perform simulations, especially Simgrid and StarPU version of Simgrid.

In last, we present various dense linear algebra libraries and matrix factorization algorithms. Most modern linear algebra libraries implement tile versions of different matrix factorizations using a runtime system, where the runtime system takes care of effective scheduling of tasks. In the past few years, while GPUs have gained in popularity, tile algorithms have heavily been employed to handle heterogeneous architectures. In that case, the runtime system may assign some tasks to the GPUs to accelerate them.

1.1 Task-based Runtime Systems

Complexity and scale of platform is increasing continuously to satisfy HPC computational needs. To cope with the increasing complexity and scale of hardware architectures and exploit the full capacity of platform with existing code, most of the computational applications are expressed at high level in the form of a DAG of tasks. Then a task scheduler or runtime system is used to schedule those tasks on the given hardware platform. In this section we provide a brief overview of different runtime systems and their important features.

1.1.1 StarPU

StarPU [22] is a runtime system developed at Inria Bordeaux, France, specifically designed for heterogeneous multicore architectures. It allows program-

mers 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 DAG, with CPU and/or GPU implementations. The code for each type of task implementation is provided separately. This separation of concerns not only allows for ensuring a modular design but it is also very convenient for writing portable codes. Indeed if one provides both CPU and GPU implementations of a task, this task can be executed on either of these units. StarPU schedules tasks and associated data transfers on available CPUs and GPUs. The data that a task manipulates is automatically transferred to the computational unit where the task has been decided to execute, so that application programmers are freed from 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 graph scheduling policies from the literature, 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. Complete description of StarPU can be found in the work by Augonnet [20].

StarPU Execution Model

In StarPU, the execution is initiated by the main thread, running on a CPU, which submits all the tasks asynchronously and the execution of tasks is performed in parallel by different worker threads (or, simply, workers). A CPU worker is bound to a CPU core while a GPU worker is bound to a GPU core and a CPU core to exploit GPU efficiently. StarPU also allows a worker to submit other tasks at runtime although it is not in the interest of sequential submission. StarPU requires registration of all data associated with a task before submitting the task. Each StarPU task contains a codelet which describes a computational kernel and its possible implementations on different architectures, such as CPU, GPU. Figure 1.1 shows an example of a StarPU codelet. It indicates on what computational units (*where* field) the corresponding task can be executed and function pointers to different implementations (cpu_funcs and gpu_funcs fields). It also indicates the number of data/handles ($nbuffers$ field) manipulated by its task. A task also describes what data are accessed and how they are accessed (read and/or write mode) during computation. Executing a task can be viewed simply as a function applying a codelet on a data set associated with the task. Task dependencies are inferred from data dependencies. However programmers are allowed to express dependencies explicitly for some data.

```
/* Codelet definition for kernel f */
struct starpu_codelet f_cl =
{
.where = STARPU_CPU | STARPU_CUDA,
.cpu_funcs = { f_cpu_func },
.cuda_funcs = { f_cuda_func },
.nbuffers = 2
};
```

Figure 1.1: A StarPU codelet.

StarPU Scheduling Model

StarPU scheduler schedules tasks when they become ready to be executed, i.e., all dependencies are satisfied. Each worker pulls tasks one by one from the scheduler. This is up to programmers how to implement a scheduler. However, StarPU provides a few schedulers based on well known dynamic task graph scheduling heuristics. All schedulers usually contain at least one queue to store tasks between the time when they become available and the time when a worker picks them. Here is the description of few StarPU schedulers which are relevant to this thesis.

- **random**: This scheduler assigns tasks randomly over all the computational resources. It uses an estimation of the relative performance of the resources to balance the randomness. This is thus representative of classical partition heuristics, which take heterogeneity of resources into account but not heterogeneity of tasks.

- **ws** (work stealing): This scheduler uses a queue of tasks per worker. All tasks released by a worker are added to its own queue. An idle worker steals a task from the most loaded worker.

- **eager**: This scheduler uses a central task queue to store ready tasks. An idle worker selects a task from the central queue. It does not give time to scheduler to prefetch data since the scheduling decisions are made very late.

- **dmda** (dequeue model data aware): This scheduler takes task execution performance models and communication models into account to make scheduling decisions. It is based on the Minimum Completion Time (MCT) heuristic [103] to assign tasks to computational resources. Each task is assigned to the worker which is expected to complete it first, taking both the estimated computation time on the estimated target

resource and the data transfers time into account, thus making it representative of the state-of-the-art HEFT heuristics [99]. Figure 1.2 exhibits working principle of **dmda** scheduler, where scheduler takes a task from global queue, a queue with set of ready tasks, and pushes it to one of the worker queues based on minimum completion time heuristic. In practice, **dmda** does not use any global queue and a ready task is directly pushed to one of the worker queues. However, for better understanding we can think that ready tasks are stored in a global queue.

- **dmdar** (dequeue model data aware ready): This scheduler is a refinement of **dmda**, where each worker picks from its queue the task whose most data is available on its associated memory.

- **dmdas** (dequeue model data aware sorted): This scheduler is another refinement of **dmda**, where tasks are sorted by priority order in each worker queue, which makes it even closer to HEFT.

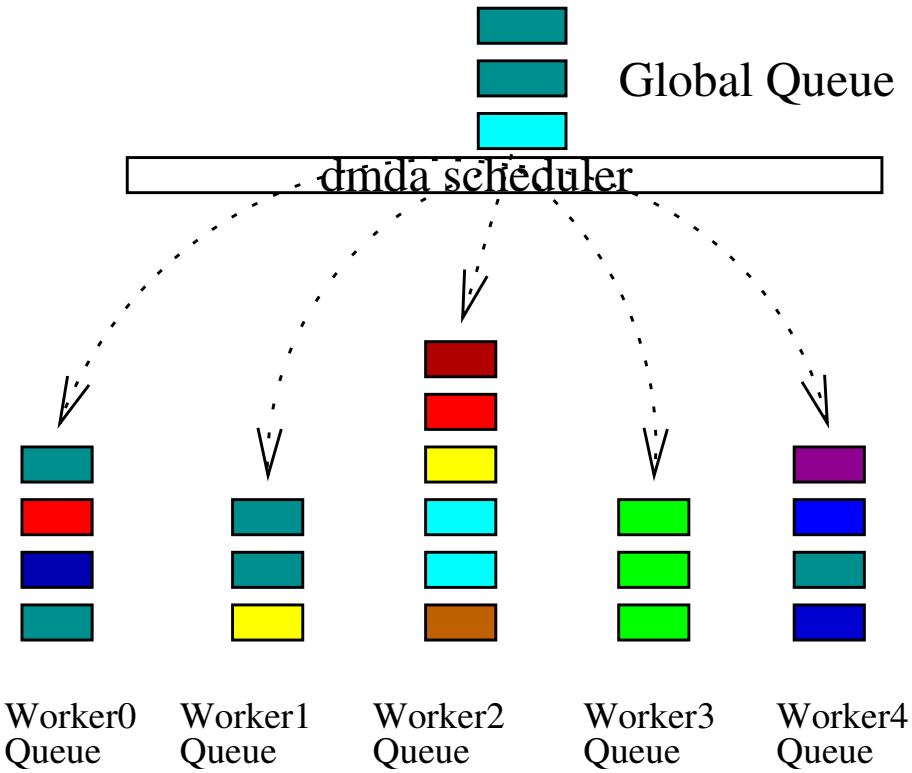


Figure 1.2: Working principle of **dmda** scheduler.

The work presented in this thesis relies on the StarPU runtime system and StarPU scheduling model. This is mostly due to its large set of features which include full control over the scheduling policy, support for hybrid platforms and efficient handling of data transfers. Here are some features of StarPU which we used extensively in this thesis.

- StarPU can generate execution traces containing execution timings of all executed tasks, data transfers information, runtime overhead and memory loads. The execution trace allows programmers to analyze the application execution in details and help to find what went wrong with the execution. This feature also helps one to develop new scheduling policies and perform different theoretical analysis such as lower bound on performance, distribution of tasks of the execution. StarPU generated trace can be visualized with different visualizer tools, such as ViTE (visual trace explorer) [4].

- Different applications exhibit different types of characteristics and therefore sometimes it is required to implement custom scheduling policies to exploit the platform in an effective way. StarPU allows programmers to implement their own scheduling policies.

- StarPU automatically calibrates and stores the execution timings of different kernels and data transfer time between two memory nodes which have not been calibrated yet. Estimation based schedulers such as **dmda** require this information in order to make scheduling decisions. We have used calibrated timings extensively to perform different simulations throughout this thesis.

- In some scenarios, one may want to force scheduling, for example force a given a set of tasks to a particular resource. StarPU provides this feature by allowing programmer to specify the worker identity while inserting the task. StarPU also allows one to specify the order in which tasks must be executed . This feature provides a total control to implement a static schedule in StarPU. In this case, at runtime it simply follows the order of execution as prescribed by the programmer. We have used this feature extensively in Chapter 2

- StarPU can use Simgrid to provide simulation support on any arbitrary platform [96]. In this case StarPU runs 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.

1.1.2 QUARK

QUeueing And Runtime for Kernels (QUARK) [104] is a runtime environment for the dynamic scheduling and execution of tasks based applications on multicore and multi-socket shared memory system, which is developed at the Innovative Computing Laboratory (ICL), University of Tennessee. It is similar to StarPU and based on the dataflow model where dependencies are inferred through a runtime analysis of data usage by the different kernels. It dynamically schedules tasks whose all incoming dependencies are satisfied. QUARK was originally designed to support dynamic linear algebra algorithms for the PLASMA linear algebra project [42]. The QUARK runtime contains several optimizations inspired by algorithms in PLASMA. It is capable to support other applications which can be decomposed into tasks with data dependencies. The goal of QUARK project is to provide an easy-to-use interface for application programmers that scales efficiently to large number of cores. QUARK does not support hybrid platforms, mainly because it does not deal with data movements.

1.1.3 PaRSEC

Parallel Runtime Scheduling and Execution Control (PaRSEC) [36] is a generic framework for dynamic scheduling of tasks on distributed many-core heterogeneous architectures, which is developed at the ICL laboratory, University of Tennessee. The high level difference between StarPU and PaRSEC is the way tasks and their dependencies are represented. In StarPU, a thread submits all tasks asynchronously and then runtime detects dependencies among different tasks. While PaRSEC uses a symbolic Parameterized Task Graph (PTG) [52] to represent tasks and their data dependencies to other tasks. PaRSEC does not build a DAG in memory and does not analyze the way tasks depend on one another by analyzing input and output data. Rather, this information is expressed by programmers in the PTG format. As a consequence, it is harder to write programs in PTG format. In PTG format, programmers have to mention all possible input and output dependencies in a compact form. The size of PTG representation for an application does not depend on the size of the problem, but only on the number of different types of tasks used by the application. As dependencies are explicitly provided by programmer, it also supports irregular applications. The PTG model is extremely scalable, the runtime can determine successors and predecessors information for any local task quickly. The DPLASMA library [35] is a dense linear algebra library implemented on top of PaRSEC runtime system.

PaRSEC has limited support for heterogeneous systems, in the case of Cholesky for instance, it only runs GEMM kernels on the GPUs, and thus uses very simple heuristics to determine which kernels to run on the GPUs.

Figure 1.3 shows PTG representation of DPOTRF kernel of Cholesky Factorization. It indicates that we have $SIZE$ instances of DPOTRF, numbered from $k = 0$ to $k = SIZE - 1$, where instance $k = 0$ of DPOTRF is immediately available, while others have to wait for the previous DSYRK tasks, and instance k of DPOTRF releases the corresponding $SIZE - k$ DTRSM tasks. Details about PTG representation and how to write program in PTG can be found in [36].

```
DPOTRF(k)
// Execution space
k = 0 . . SIZE−1
//Flows and dependencies
T <− (k == 0) ? A(k, k) : T DSYRK(k−1, k)
−> T DTRSM(k, k + 1 . . SIZE−1)
−> A(k, k)
```

Figure 1.3: PTG representation of POTRF kernel.

PaRSEC provides a language called Job Data Flow (JDF) to express PTG parallel codes. Then a specific compiler, known as *daguepp*, converts JDF code to C-code.

1.1.4 OpenMP

Due to the wide popularity of task based runtime systems, tasking feature was included in OpenMP version 3.0 [80]. Tasking facilitates the parallelization of applications where tasks are created in a recursive way or through a while loop. An explicit task is specified using the task directive. The task directive defines the code associated with the task and its data environment. A thread creates a new task after encountering a task construct. The task construct can be placed anywhere in the program. Note that OpenMP is not really a runtime system, but an interface standard, implemented by various systems, notably compilers. It however guides a lot how the underlying runtime works.

When a thread creates a task, it may defer the execution of the task for later. If task execution is deferred, then the task is placed in a pool of tasks. The threads in the current team will take tasks from the pool and execute them until the pool is empty. The thread that executes a task may be different from the thread that originally submitted it.

By default, tasks are tied to the thread that first executes them, it may not be the creator thread. Programmers can use "untied" clause to remove all restrictions. "untied" tasks provide more freedom to implementation and can be scheduled based on different heuristics, such as load balancing.

The task construct was extended with a depend clause in OpenMP version 4.0 [81], which enables OpenMP runtime to automatically detect dependencies among tasks and schedule them appropriately. The same OpenMP version also provides support for accelerators with a target construct. The execution model of OpenMP for accelerators is host centric and it assumes that each accelerator device is attached to a host device. A target region begins as a single thread execution and when a target construct is encountered, the implicit device thread executes the target region and the encountering thread waits at the construct until the execution of region completes.

OpenMP does not provide any freedom to the runtime to decide on which CPU or GPU to run tasks. However, with a few extensions to OpenMP such as OpenMP interface of StarPU [14] (which does not support hybrid platforms yet), we could relax it and improve performance of OpenMP applications.

1.1.5 StarSs

StarSs [82] is a sequential task-based programming model, developed at Barcelona Supercomputing Center (BSC), where programmer writes sequential code in a traditional programming language (i.e., C, C++ or Fortran) which is executed in parallel and runtime manages the data dependencies and data movements between tasks. Many instantiations of StarSs have been developed to target different architectures: CellSs for Cell processors, SMPSs for shared memory machine and homogeneous multicore processors, GridSs for computational grids, GPUSs for heterogeneous multi-accelerator platforms. In StarSs, user annotates the applications to target a particular architecture. It uses a few OpenMP like pragmas to identify tasks in the user code. It implements data renaming to eliminate false dependencies. StarSs implements a task hierarchy which allows instantiation of subtasks within a task in the following way. Each task creates a private context for its subtasks. Synchronization and data dependencies are considered in the same context. A given task waits for the end of its children tasks before finishing.

Planas et al. have exhibited some experiments that combine a first task level with SMPSs and a second task level with CellSs to take advantage of both architectures [82]. Tasks are scheduled in a hierarchical fashion, however CellSs tasks are only scheduled once the corresponding SMPSs task has been assigned to a Cell processor. Therefore, having separate runtime systems does not allow to actually schedule tasks between heterogeneous types of processing units unless the programmer explicitly selects the target platform, and therefore which runtime system should process the task.

The main difference between StarPU and the different instantiations of StarSs is that StarPU really provides a unified abstraction of driver that makes it possible to deal with different types of platforms. In the StarSs programming model, execution timings are not known in advance which prevents it from

implementing estimation based strategies, such as HEFT.

OmpSs is an attempt to integrate features from the StarSs programming model in to a single programming model [57]. It extends OpenMP with new directives to support asynchronous parallelism and hybrid architectures. Similar to StarPU, in OmpSs user writes code for a single address space which may execute in several non coherent address spaces. OmpSs implements data packing to minimize the number of transfers among different memories. It uses locality aware work stealing strategies to achieve load balancing among different processing units. The OmpSs programming model presently supports the following architectures: 1) Intel 32- and 64-bit platforms including support for CUDA on NVidia GPUs, 2) Intel MIC in native and offload modes, 3) IBM Power8 platforms including support for CUDA on Nvidia GPUs, and 4) ARM 32- and 64-bit platforms, including support for OpenCL MALI GPUs (32 bits version only).

1.1.6 XKaapi

XKaapi [67] is a runtime system for data flow programming on multi-CPU and multi-GPU architectures developed at Inria Grenoble, France. It provides different APIs to program heterogeneous parallel architectures in C, C++ and Fortran. It relies on different work stealing heuristics to ensure load balancing among different processing units. It uses fully asynchronous task execution strategy on GPUs to overlap computations with data transfers. It creates a system thread and a work queue for each computational resource. The unique feature of XKaapi is that it minimizes the overhead of critical paths by postponing the data dependencies computations to idle threads. Therefore, it moves the cost of computing ready tasks from task's creations to the steal operations performed by idle threads. It works on the following work stealing mechanism: an idle thread submits a steal request to a randomly chosen victim. On reply, the requesting thread gets a copy of one ready task and original task is marked stolen in victim's queue. To find a ready task, the requesting thread iterates through the victim's queue from the least recent pushed task to the most recent one and computes true data dependencies for each task. The iteration stops when requesting thread finds a ready task in the victim's queue. XKaapi also uses two different locality aware heuristics with work stealing to improve the performance. The first one is based on minimization of data transfers among resources during steal operation. A task is assigned to the resource which owns the largest sum of input bytes. The second heuristic is based on an owner compute rule, a task is assigned to the resource which minimizes the number of invalidation of data replicas. Ties are broken by selecting a resource randomly among the set of eligible resources. Both heuristics actually do not correspond to steal mechanism and push tasks to remote workers. XKaapi also implements different queues for each worker to accelerate the search operation

for a ready task while stealing.

Similar to Cilk [32] and OmpSs, in XKaapi, a task can create children tasks which is not the case with other data flow programming libraries, such as StarPU, QUARK. False dependencies can be eliminated in XKaapi through variable renaming by using extra memory and write back policy is used to maintain the data coherency. Similar to StarPU, XKaapi uses codelet based low overhead task representations that allow to handle a high degree of parallelism efficiently. A task may have multiple implementations, such as a CPU implementation and a GPU implementation. At least one implementation is required for each task. The implementation may be recursive, which allows XKaapi to decompose some tasks further to subtasks operating on smaller data.

Most recent GPUs, such as Fermi have one execution engine and two copy engines, which enable to perform a kernel execution and two way memory transfers simultaneously. Similar to other runtimes, XKaapi also takes advantage of this by using a new data structure, called kstream, which combines together three types of CUDA streams: a stream for host to device transfer, a stream for kernel execution and a stream for device to host transfer.

1.1.7 SuperMatrix

SuperMatrix is a multithreaded runtime system that parallelizes matrix operations for SMPs and multi-core architectures [46]. It views matrices hierarchically, matrices of matrices. The unit of computation is operations on a single submatrix. It enqueues the required operations, tracks dependencies, and then executes the operations utilizing out-of-order execution techniques inspired by superscalar processors.

The Formal Linear Algebra Methods Environment (FLAME) project uses SuperMatrix runtime system to parallelize dense and carefully structured sparse linear algebra computations [69]. Detecting dependencies across different iterations is very difficult in FLAME, as different submatix views may reference to the same block. It requires complete knowledge of matrix partitioning to determine what regions of matrix are being referenced. However, FLASH (Formal Linear Algebra Scalable Hierarchical), extension of FLAME API, delimits the block referenced by each submatrix view.

API for defining tasks on OmpSs and SuperMatrix is quite different. OmpSs uses annotations, similar to OpenMP, which are placed around function calls to denote different tasks. Then, a source to source compiler converts these annotations to code that performs dependency analysis and out of order execution. The load balancing of tasks can vary based on the computational runtime of each function. While in SuperMatrix, computation runtime of each task depends on the size of each submatrix created using the FLASH API.

1.1.8 Legion

Legion is a data centric programming model and runtime system for achieving high performance on distributed heterogeneous architectures developed at Stanford University [25]. It provides an interface such that programmers can explicitly declare different properties of program data, such as data organization, partitioning. It also allows programmers to control the mapping of tasks on to different architectures. It uses logical regions to describe the locality and independence of data. Each Legion program executes as a tree of tasks with a top level task generating sub-tasks which can recursively generate further subtasks. It provides the ability to partition data in multiple ways and to migrate data dynamically between these views as application moves between different phases of computation.

The Legion programming model uses a *software out-of-order processor*, or SOOP, for scheduling tasks. The SOOP takes locality and independence properties captured by logical regions into account while making scheduling decisions.

1.2 Simulation Framework

In recent years, advances in hardware and software technologies made it possible to execute different HPC applications over increasingly large sets of resources. The study of scheduling problems for such applications and platforms has been quite significant in recent times. Simulation is a popular and effective way to evaluate and compare different scheduling algorithms over a wide range of scenarios.

Many fine grained simulators such as GPGPU-Sim [23] have been developed for GPUs in past years which simulate at cycle level. There are also a few GPU-specific simulators such as Barra [51] for the Nvidia G80, Multi2Sim [100] for the AMD Evergreen GPU. Simulation time of these simulators is very long because every detail of the specific GPU is simulated. There are a few simulators such as SST [90], TaskSim [88] which are based on multiple levels of abstraction to provide good prediction. However these address only multicore machines with no GPUs so far. We use Simgird [45] simulator in this thesis which is accurate enough for our need while being very fast.

1.2.1 Simgrid Simulation Engine

Simgrid is a versatile simulation toolkit initially designed to study the behavior of different scheduling algorithms on 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 [96] 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 [21]. 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 [96], and properly reproduce the various behaviors that can be observed for the different 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 performed. 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 experiments for this thesis, since the platform became unavailable for a couple of times due to different issues such as air conditioning, software upgradation, transition from PBS to SLURM job scheduler.

- The platform can be modified, for instance to change the available PCI bandwidth, the execution times of the kernels, etc. In Chapter 2, we use this feature in order to build a virtual "related" heterogeneous platform.

Simgrid version of StarPU allows to perform simulations on any machine by using the configuration files of target platform and expected execution time of kernels on each resource of the target platform. We use this to evaluate the effectiveness of the different scheduling algorithms on a single heterogeneous node in Chapter 2.

Simgrid does not provide a framework to support simulation within a simulation (two levels of simulations) and to handle spoliation of tasks. These two features were required to evaluate some of our scheduling algorithms. Therefore, we have written our own simulator to support both of these features and used it in Chapters 3, 4 and 5.

1.3 Dense Linear Algebra Libraries

Linear systems of equations, Least squares problems, Eigen value problems and Singular value decomposition problems are the basic problems of linear algebra. In this section, we briefly describe some old and some state-of-the-art numerical linear algebra libraries designed for dense matrices.

1.3.1 LINPACK (LINear algebra PACKage)

LINPACK is a software library written in FORTRAN66 by Jack Dongarra, Jim Bunch, Cleve Moler, and Gilbert Stewart [1]. This project had started in 1974 and it was intended for use on supercomputers in the 1970s and early 1980s. During that period, supercomputers with vector processors were very popular, therefore LINPACK was basically designed to exploit vector processors. This library provides routines to solve systems of linear equations for general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square matrices. It also provides routines to compute QR and singular value decompositions of rectangular matrices. It makes use of BLAS libraries for performing vector operations (BLAS level 1 operations). LINPACK has been largely superseded by LAPACK [19], which has been designed to run efficiently on machines with hierarchical memory design.

1.3.2 LAPACK (Linear Algebra PACKage)

LAPACK [19] is a standard software library for numerical linear algebra. It was originally written in FORTRAN77, but moved to FORTRAN90 in version 3.2 (2008). It provides routines for solving systems of linear equations, least square solutions of systems of linear equations, eigen value problems and singular value problems. It also provides routines to implement different matrix factorizations such as LU, Cholesky, QR, SVD and Schur. It provides routines for both real and complex matrices in both single and double precision. It handles dense and banded matrices, but not general sparse matrices. LAPACK library was first released in 1992.

The original goal of LAPACK project was to make LINPACK and EISPACK [59] libraries to run efficiently on shared memory vector and parallel processors. On these machines LINPACK and EISPACK are inefficient because memory access patterns do not take multi layer memory hierarchies into account, therefore spending too much time moving data instead of doing useful floating-point computations. LAPACK solves this problem by reorganizing the algorithms to use block matrix operations such as matrix multiplication in the innermost loop. These block operations can be optimized for each architecture to account for memory hierarchy and so provide a portable way to achieve high efficiency on different modern machines.

Maximum efficacy of LAPACK routines are performed by calls to different Basic Linear Algebra Subprograms (BLAS). LAPACK is designed to exploit BLAS level 3 operations. Coarse granularity of BLAS level 3 operations assists to obtain high efficiency on many high performance computers.

LAPACK uses multi threaded implementation of BLAS libraries to efficiently exploit SMP processors. LAPACK has also been extended to run on distributed memory system in later packages such as ScaLAPACK [28] and PLAPACK [18].

1.3.3 PLASMA (Parallel Linear Algebra Software for Multicore Architectures)

The main goal of the PLASMA project is to address performance shortcomings of LAPACK and ScaLAPACK libraries on multicore processors and multi-socket systems of multicore processors [42]. PLASMA uses tile based data layout and provides implementation of state-of-the-art algorithms using task based scheduling techniques. It assigns work to cores based on the availability of data for processing at any given point during execution. It is based on data driven scheduling, which is close to the idea of Section 1.1, where computations are expressed through a DAG, and DAG is explored at runtime. PLASMA uses QUARK runtime system to perform dynamic scheduling of tasks.

PLASMA has been designed to supersede LAPACK and ScaLAPACK by restructuring the software to expose more parallelism and achieve much greater efficiency, where possible, on modern computers based on multicore architectures. It also relies on new or improved algorithms. PLASMA does not replace ScaLAPACK as software for distributed memory computers, since it only supports shared memory machines.

PLASMA has also been extended to run on distributed memory system in later package DPLASMA [35].

1.3.4 MAGMA (Matrix Algebra for GPU and Multicore Architectures)

MAGMA library is an extension of LAPACK library for GPU and multicore architectures [98]. It uses static scheduler for distribution of work on different computational units. It schedules embarrassingly parallel tasks such as GEMM on GPU and small tasks which are very less parallelizable and often on critical path such as POTRF on CPU. In MAGMA, algorithms are split of varying granularity to utilize different hybrid component efficiently. It also supports out-of-device memory algorithms by dividing the matrix into different sub-matrices and transferring a submatrix to GPU to perform computations and then remaining matrix is updated accordingly. It uses 1-D block cyclic

data distribution [97] to support multiple GPUs. It handles real and complex matrices in both single and double precisions.

The goal of MAGMA is to design linear algebra algorithms and frameworks for hybrid multicore and multiGPU systems that can enable applications to fully exploit the power of each hybrid component. We performed performance comparison of some of our approaches with MAGMA in Chapter 4.

1.3.5 MORSE (Matrices Over Runtime Systems at Exascale)

To cope with the increased degree of parallelism, a new class of linear algebra algorithms has been proposed, often referred as tile algorithms in the literature [44, 85]. These algorithms led to the design of new libraries in the past five years such as PLASMA, FLAME and DPLASMA. 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, Supermatrix, PaRSEC) 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 main goal of MORSE [5, 6] project is to enable different numerical algorithms to execute on a scalable unified runtime system which exploits the full potential of future exascale machines. To develop numerical linear algebra softwares that will perform well on petascale and exascale systems with thousands of nodes and millions of cores, several challenges have to be overcome, both by numerical linear algebra and runtime system communities. MORSE project aims at describing linear algebra algorithms at a high level of abstraction, which will enable the strong collaboration between linear algebra, runtime system, and scheduling communities to fully benefit from the potential of future large scale machines. This project aims at bridging the immense software gap that has opened up in front of the HPC community.

1.3.6 CHAMELEON

Chameleon is a sub-project of MORSE specifically dedicated to dense linear algebra [8]. It relies on sequential task-based algorithms where tasks of the algorithms are submitted to a runtime system. Such a system is a bridge between the application and the hardware which handles the scheduling, data transfers and the effective execution of tasks on to the processing units. A

runtime system such as StarPU is able to manage automatically data transfers between non-shared memory areas (CPUs-GPUs, distributed nodes). This kind of implementation paradigm allows to design high performing linear algebra algorithms on very different types of architectures.

The Chameleon library is based on the PLASMA tile 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 is that, when aiming at analyzing different scheduling strategies, it allows to run in simulation mode with the field-proven combination [96] of StarPU and Simgrid. Chameleon also supports PaRSEC, QUARK and OmpSs runtime systems.

1.4 Dense Matrix Factorizations

Dense matrix factorizations are the basis of many scientific applications. In this thesis we consider one of them, namely the Cholesky factorization, extensively for our experiments. We also consider the QR and LU factorizations for some of our experiments. In this section, we briefly describe the Cholesky, QR and LU factorizations.

1.4.1 Cholesky Factorization

The Cholesky factorization (or Cholesky decomposition) decomposes a positive definite matrix A into a unique lower triangular matrix L such that $A = LL^\top$. This type of factorization is very useful for efficient numerical solutions and different types of simulations such as weather predictions [47], Monte carlo simulations [66] and optics simulations [74]. It involves around $\frac{N^3}{3}$ floating point operations for $N \times N$ matrix. Where applicable, it is almost twice more efficient than the LU factorization for solving systems of linear equations. The Cholesky factorization also decomposes a positive semi definite matrix into a lower triangular L such that $A = LL^\top$, but such decomposition may not be unique.

Here is an example of the Cholesky factorization for a positive definite matrix.

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 20 & 26 \\ 3 & 26 & 70 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 4 & 0 \\ 3 & 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$

The Cholesky factorization 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. $Ax = b$ can

be solved by first computing a lower triangular matrix L such that $A = LL^\top$, then solving $Ly = b$ for y by forward substitution and finally solving $L^\top x = y$ for x by backward substitution.

Linear systems of equations $Ax = b$ often arise in physics applications, especially when looking for numerical solutions of partial differential equations or solving least square problems, where A is positive-definite due to the nature of the modeled physical phenomenon [62]. There is abundant literature on the implementation of dense as well as sparse Cholesky factorization on different platforms [86, 87, 79, 92, 91, 49].

Tile Cholesky Factorization

To take advantage of modern highly parallel architectures, state-of-the-art numerical algebra libraries implement tile Cholesky factorizations. The matrix $A = (A_{ij})_{0 \le i,j \le N}$ is divided into $N \times N$ tiles (or blocks) of $N_b \times N_b$ elements, and the tile 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 [65, 44, 85]. Following the BLAS and LAPACK terminology, the tile algorithm for Cholesky factorization is based on POTRF, TRSM, SYRK, and GEMM kernel subroutines.

Algorithm 2 shows the pseudo-code of the tile version of the Cholesky factorization implemented in the Chameleon library. In each instance of the outer loop, a Cholesky factorization (POTRF kernel) on the ith diagonal tile is performed and the trailing panel is updated with triangular solve (TRSM kernel). Then, the remaining trailing submatrix is updated by applying symmetric rank-k updates (SYRK kernel) on the diagonal tiles and general matrix multiplications (GEMM kernel) on non-diagonal tiles.

Algorithm 1: Tile Cholesky Factorization.

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

The sequence of computations for tile Cholesky factorization can be represented with a DAG of tasks as depicted in Figure 1.4 in the case of 5×5 tile matrix.

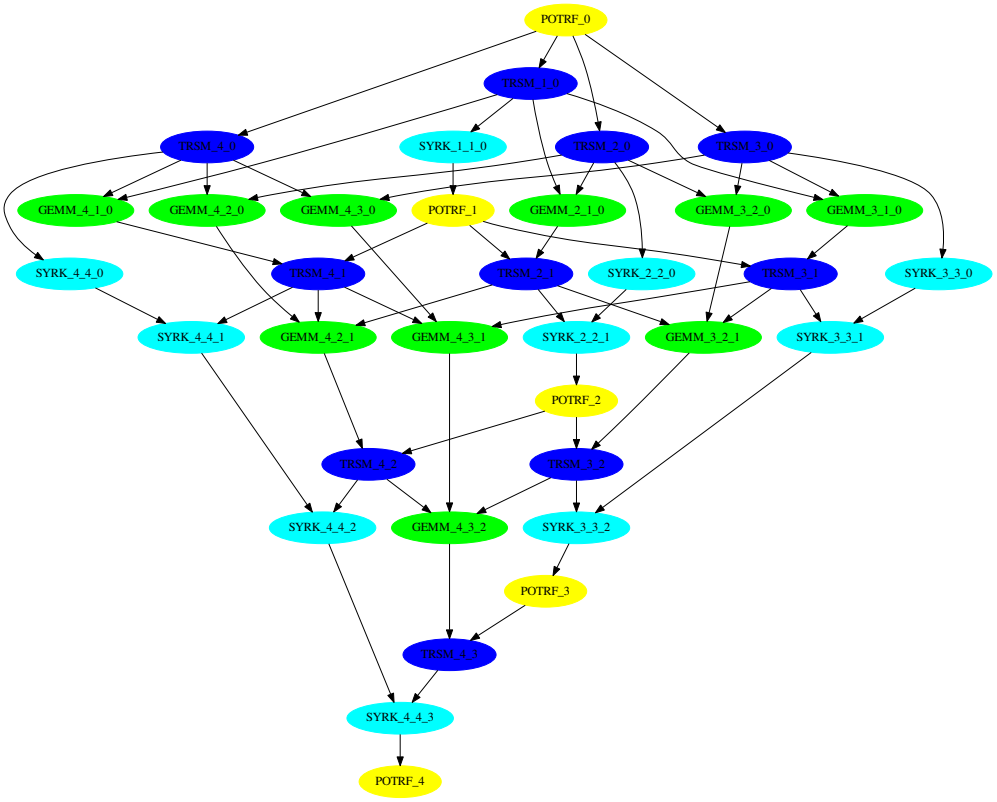


Figure 1.4: DAG of the tile Cholesky factorization - 5×5 tile matrix.

The Cholesky factorization task graph has $\frac{N(N+1)(N+2)}{6}$ vertices and $\frac{(N-1)N(N+1)}{2}$ edges for $N \times N$ tile matrix, more precisely, it has N POTRF, $\frac{N(N-1)}{2}$ TRSM, $\frac{N(N-1)}{2}$ SYRK and $\frac{N(N-1)(N-2)}{6}$ GEMM tasks.

Figure 1.5 depicts performance of tile Cholesky factorization in actual execution and simulation on a heterogeneous node with 9 CPU computing cores of X5650 processor and 3 Nvidia Tesla M2070 GPUs. We use StarPU runtime system for the actual execution and Simgrid version of StarPU for the simulation. In all cases we use HEFT based StarPU scheduler for scheduling of tasks. In "Simulation with no communication" mode, we modify the platform files of the machine and set all transfer associated costs to zero. Figure 1.5 shows that the simulated performance is slightly better than the actual execution performance, which can be explained with the fact that actual execution suffers from scheduling overhead. Simulated performance in both cases, with and without communication costs, is comparable. Therefore we can say that

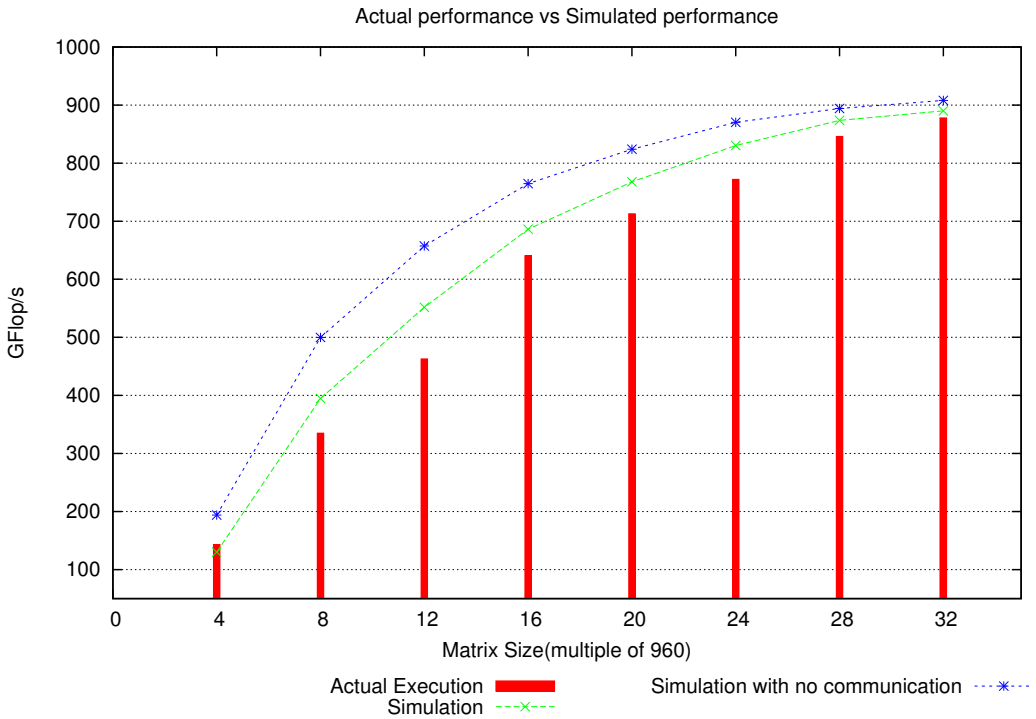


Figure 1.5: Impact of communication on performance of Cholesky factorization.

kernel	PORTF	TRSM	SYRK	GEMM
CPU time/GPU time	≃2.3	≃11	≃26	≃29

Table 1.1: GPU acceleration ratio over CPU core for Cholesky tasks (size 960).

communication costs do not impact the performance of Cholesky much and Cholesky is a compute intensive application.

Table 3.1 depicts the acceleration ratios of different Cholesky tasks on a GPU (Nvidia Tesla M2070) over a CPU core (one core of Intel Xeon X5650 processor) for a tile size of $N_b = 960$. Different Cholesky tasks exhibit strongly heterogeneous and unrelated acceleration ratios: GPUs are for instance much more efficient to process regular kernels such as matrix-matrix multiply (GEMM) rather than more irregular kernels such as matrix factorization (POTRF). Cholesky factorization is also showing a complex pattern of data dependencies, where parallelism increases and then decreases as execution progresses. It is crucial for the scheduler to make efficient utilization of all resources when the number of ready tasks is large and make better choice of resources when few tasks are ready. Due to the above reasons, Cholesky factorization is an ideal candidate for the study of task based scheduling on heterogeneous architectures.

We use the Cholesky factorization extensively for our experiments in this thesis. We consider Chameleon implementation of the tile Cholesky factorization on top of StarPU runtime system for the actual execution and Cholesky task graphs generated by Chameleon with StarPU runtime system for the theoretical analysis.

1.4.2 QR Factorization

The QR factorization (or QR decomposition) decomposes a matrix A into a product $A = QR$ of an orthogonal matrix Q and an upper triangular matrix R. It is mainly used to solve systems of linear equations and linear least squares problems.

There are several ways to compute a QR factorization such as Gram-Schmidt process, Householder transformations. In the case of Gram-Schmidt process, column vectors of matrix A are converted to the set of orthogonal vectors Q and transformations applied on the column vectors to obtain Q are represented with an upper triangular matrix R such that $A = QR$. In the presence of rounding errors on a computer, this process, also known as Classical Gram-Schmidt, is numerically unstable and Q quickly losses its orthogonality. A more stable variant of this process, known as Modified Gram-Schmidt, ensures that impact of rounding errors on orthogonality of Q is minimal. However, the computed Q is guaranteed to be nearly orthogonal only for well conditioned matrices [68]. On the other hand, in Householder transformation, a plane or hyperplane is used to reflect a vector such that all coordinates except one disappear. This method is more stable than Gram-Schmidt process. Different dense linear algebra libraries such as LINPACK [1], LAPACK [19], PLASMA [42], Chameleon [8] implement the QR factorization based on Householder transformations.

Here is an example of the QR factorization.

$$\begin{pmatrix} 6 & -3 \\ 8 & -4 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} \frac{3}{5} & 0 \\ \frac{4}{5} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 10 & -5 \\ 0 & 2 \end{pmatrix}$$

To take full advantage of modern architectures, recent linear algebra libraries implement the tile version of the QR factorization [43]. We use Chameleon implementation of tile QR factorization running on top of StarPU runtime system for our experiments. Chameleon implements tile QR factorization with four different types of kernels, namely *GEQRT*, *ORMQR*, *TSQRT* and *TSMQR*. These kernels are explained in details in [43, 12]. Figure 1.6 depicts the task graph for the tile QR factorization of a 4×4 tile matrix.

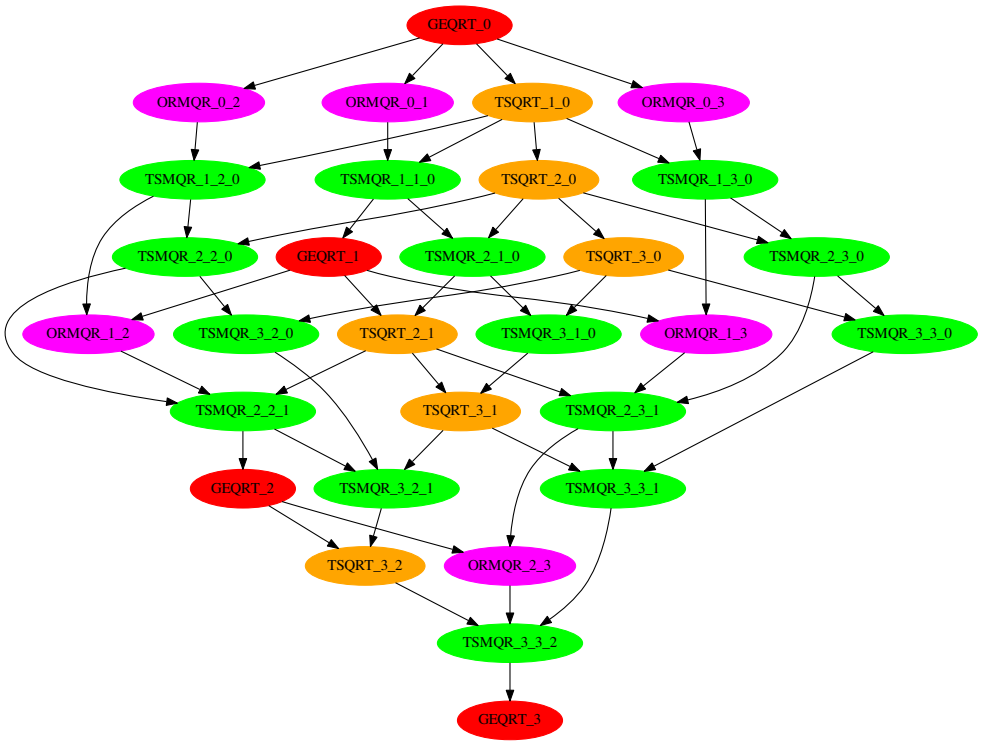


Figure 1.6: DAG of the tile QR factorization - 4×4 tile matrix.

1.4.3 LU Factorization

The LU factorization (or LU decomposition) decomposes a matrix A into a product $A = LU$ of a lower triangular matrix L and an upper triangular matrix U. A permutation matrix is sometimes used in the product as well. It can be viewed as the matrix form of Gaussian elimination. It is mainly used to solve systems of linear equations (similar to Cholesky, first solve for lower triangular matrix by forward substitution and then solve for upper triangular matrix by backward substitution). It is also used to compute inverse and determinant of a matrix. Alan Turing first introduced the LU factorization in 1948.

Here is an example of the LU factorization.

$$\begin{pmatrix} 1 & 1 & -2 \\ 2 & 14 & 4 \\ 3 & 18 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 4 & 0 \\ 3 & 5 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & -2 \\ 0 & 3 & 2 \\ 0 & 0 & 5 \end{pmatrix}$$

Most linear algebra libraries implement the tile version of the LU factorization to fully exploit the potential of modern architectures. We consider task graphs of tile LU factorization produced by Chameleon library running on top of StarPU runtime system for our simulations in Chapter 5.

We have used different state-of-the-art software libraries to perform our experiments. In Chapters 2, 3, and 4, we conducted actual executions with Chameleon library running on top of StarPU. In Chapter 2, we also used Simgrid version of StarPU to perform simulations with Chameleon library. We used different task graphs, obtained by running Chameleon library with StarPU runtime system, to perform simulations in Chapters 3, 4 and 5.

Chapter 2

Performance and Bounds of Cholesky Factorization

We consider the problem of allocating and scheduling Cholesky factorization on fully heterogeneous platforms made of CPUs and GPUs. 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 chapter 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.

2.1 Introduction

Our objective is to optimize the performance of Cholesky factorization on a hybrid computing platform. As mentioned in Chapter 1, to take full advantage of modern hybrid architectures, most linear algebra applications are expressed as task graphs at high level and then a runtime system is used to perform scheduling of tasks onto computing resources and 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 even on homogeneous platforms [60]. 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 [63].

Several scheduling heuristics have also been proposed for heterogeneous platforms, and among them the best-known certainly is heterogeneous early finish time (HEFT) [99], which inspired some dynamic scheduling strategies used in state-of-the-art runtimes. However, a large gap remains 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 are not so many GEMMs compared to other less efficient tasks, that is why there is still a large gap between GEMM peak performance and the best-achievable Cholesky performance.

In this chapter, we optimize the dynamic scheduling of the Cholesky factorization of a dense, symmetric, and positive-definite double-precision matrix, using one runtime system, StarPU, and provide better makespan bounds to prove the quality of our schedules. The contributions of this chapter 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 factorizations.

2.2 Context

2.2.1 Cholesky Factorization

Figure 2.1 depicts the task graph for the tile Cholesky factorization of a 5×5 tile matrix. We refer the interested reader to Chapter 1.4.1 for more details on the tile version of the Cholesky factorization.

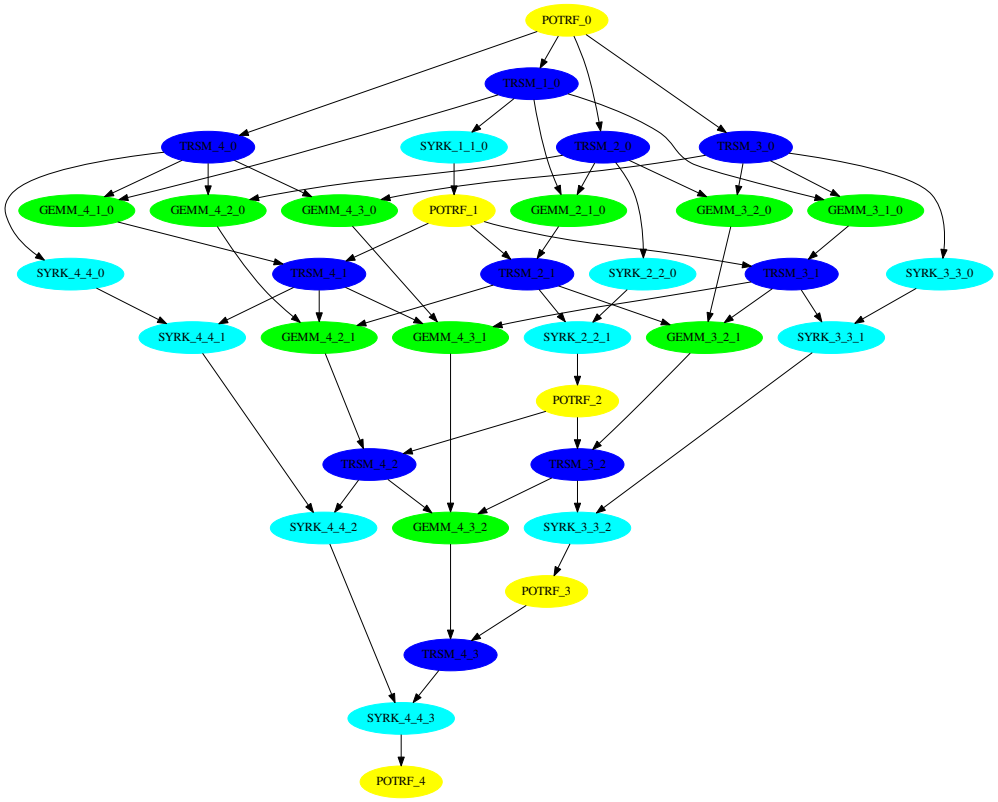


Figure 2.1: Task graph of the Cholesky factorization – 5×5 tile matrix.

2.2.2 Multiprocessor Scheduling

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 [54] 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 [72]. It ensures a good load and memory usage balancing for homogeneous computing resources [54]. 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 [42] library provides an option for relying on such static schedules on multicore chips.

Dynamic Task Graph Scheduling

Dynamic strategies have been developed in order to design methods that are 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 [48, 75]. 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 time can be very hard, thus justifying the design of ad-hoc tools such as StarPU.

As presented earlier, many heuristics have been proposed for scheduling 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 [99]. 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 [93].

2.3 Makespan Lower Bounds

Performance results for linear algebra computations 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). Indeed this is the most efficient dense linear algebra operation, and thus provides 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

GEMM tasks but much less efficient tasks such as POTRF, 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.

2.3.1 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 Chapter 1.1.1) 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:

minimize the makespan l such that

$\forall t$, all N_t tasks of type t get executed over the various processing element types r:

$$\sum_r n_{rt} = N_t$$

$\forall r$, the M_r resources of type r complete all their tasks of various types t within the makespan l:

$$\sum_t n_{rt} T_{rt} \leq l \times M_r$$

$\forall r, t$ $n_{rt} \in \mathbb{N}^+$

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 $\frac{n(n-1)}{2}$ TRSM tasks, and $n-1$ of the $\frac{n(n-1)}{2}$ 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. The resulting lower bound is called the *mixed bound* in the rest of this chapter. 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.

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.

2.3.2 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:

minimize l such that

$\forall i$, only one type of resource executes task i:
$\text{OnlyOne}(b_{i1}, \ldots, b_{iR})$

$\forall i$, task i completes:
$s_i + \sum_r b_{ir} T_{ir} \leq l$

$\forall r, \forall t$, at time θ the M_r resources of type r are executing at most M_r tasks:
$|\{i \text{ st } s_i \leq \theta < s_i + \sum_r b_{ir} T_{ir}\}| \leq M_r$

$\forall i \to j$, dependency $i \to j$ is respected:
$s_i + \sum_r b_{ir} T_{ir} \leq s_j$

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 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, 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. 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 our interest.

2.3.3 Upper bounds on performance

Lower bounds on execution time also provide upper bounds on the performance. Therefore, we have plotted different theoretical performance upper

bounds of the Cholesky factorization in Figure 2.2, based on real execution timings of different tasks on the Mirage machine (described in Section 2.4.2).

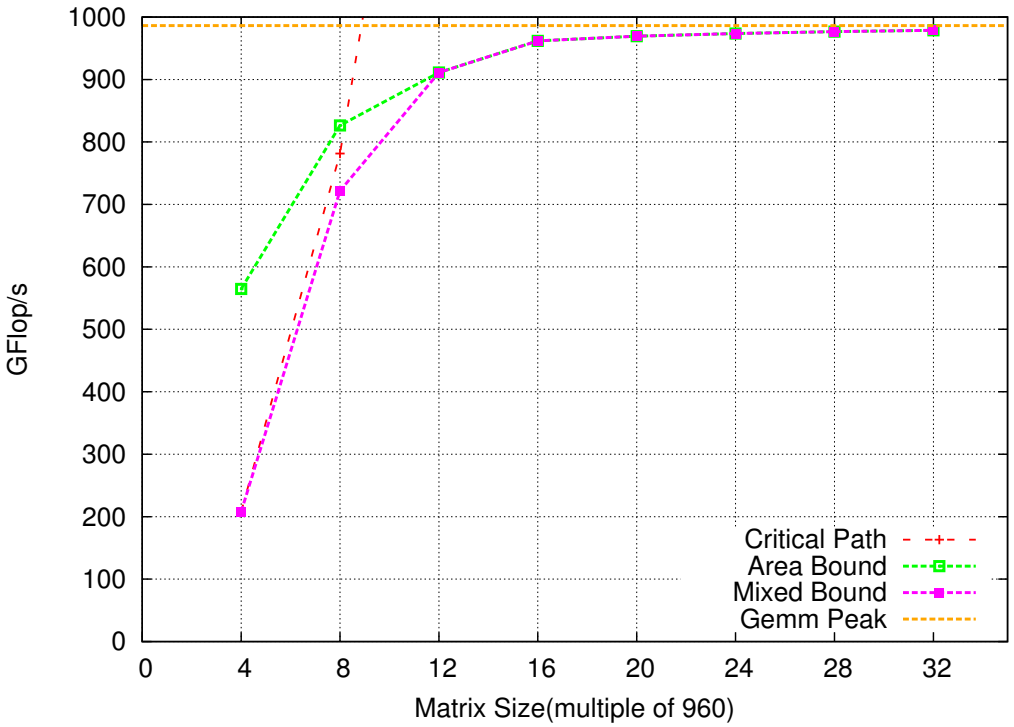


Figure 2.2: Heterogeneous theoretical performance upper bounds.

The critical path bound is calculated based on the critical path of the Cholesky task graph. While calculating the critical path, we considered 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 Section 2.3.1. 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 2.3.2 will be discussed in Section 2.4.3.

2.4 Experiments and Results

For this study, we used the Chameleon [8] implementation of the Cholesky factorization, running on top of the StarPU runtime system. We performed

actual executions on the target platform, and we additionally used the Simgrid [45, 96] simulator, in order to reduce the experimentation time, improve reproducibility of the experiments, and also be able to modify the execution platform. 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.

In "actual execution mode", we perform the real execution on Mirage machine (described in Section 2.4.2) 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 the target platform and expected execution times of kernels on each resource of the target platform.

2.4.1 Schedulers

We have experimented with a few schedulers of StarPU, namely *random*, *dmda* and *dmdas*, which are representative of state-of-the-art dynamic heuristics. The *random* scheduler assigns tasks randomly over all the computation resources. 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. 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 [99, 22]. More details about these schedulers are present in Chapter 1.1.1.

We are computing 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 this extra information as static knowledge, to influence the scheduling decisions and achieve better performance.

2.4.2 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.

2.4.3 Results

We have divided our experiments into two categories based on the types of configurations 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 know that good performance in heterogeneous case is achieved on our platform with a tile size of 960 [13, 12], 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.

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 2.3, it is clear that the random scheduler does not perform well. This happens because it does not take already assigned workload of the workers into account while making scheduling decisions and 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 2.3 also shows that dmdas slightly under-performs compared to dmda for small matrices. This 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

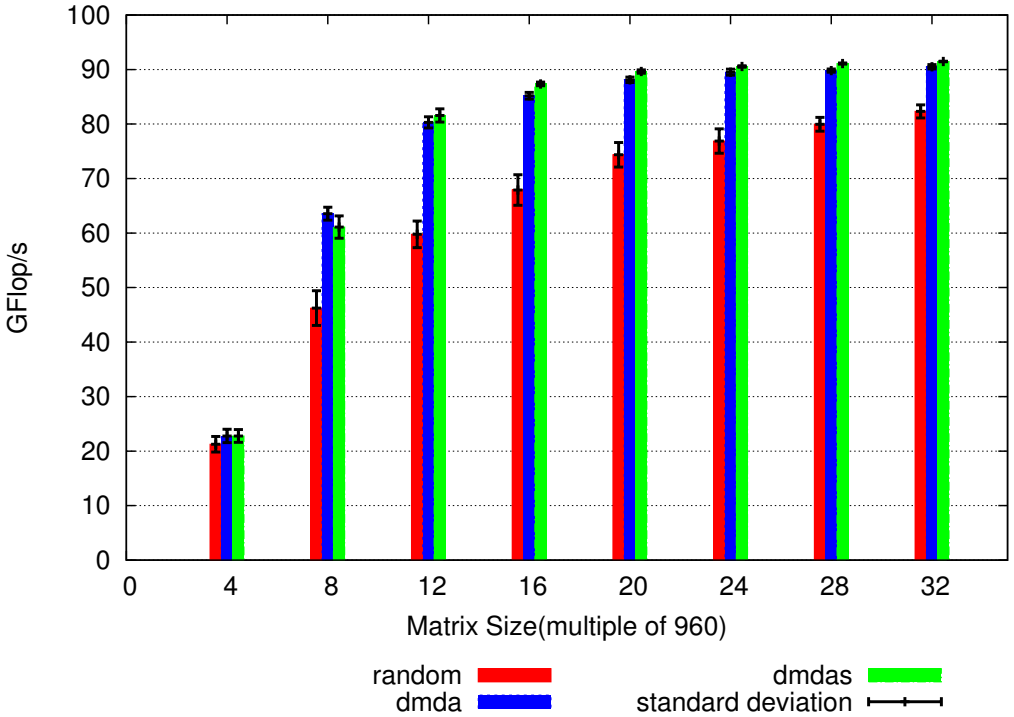


Figure 2.3: Homogeneous actual execution performance.

parallelism. But as time progresses, *dmdas* selects tasks which release a higher number of tasks, because these tasks would be the critical ones.

We are also interested to know the gap between performance of considered schedulers and upper bound. Since actual executions add some runtime overhead and affect the performance, to mitigate this overhead we have compared the bound with simulated performance.

Figure 2.4 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.

Heterogeneous Case

In this section, 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 2.1 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

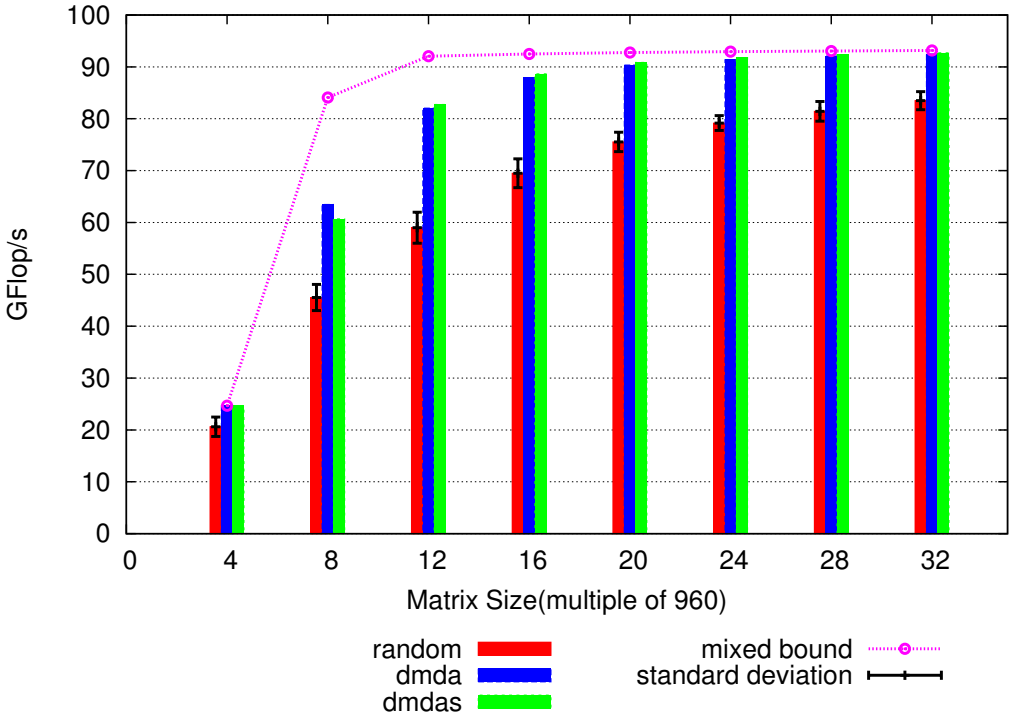


Figure 2.4: Homogeneous simulated performance.

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

Table 2.1: GPUs relative performance.

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 acceleration 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 2.3.1) 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).

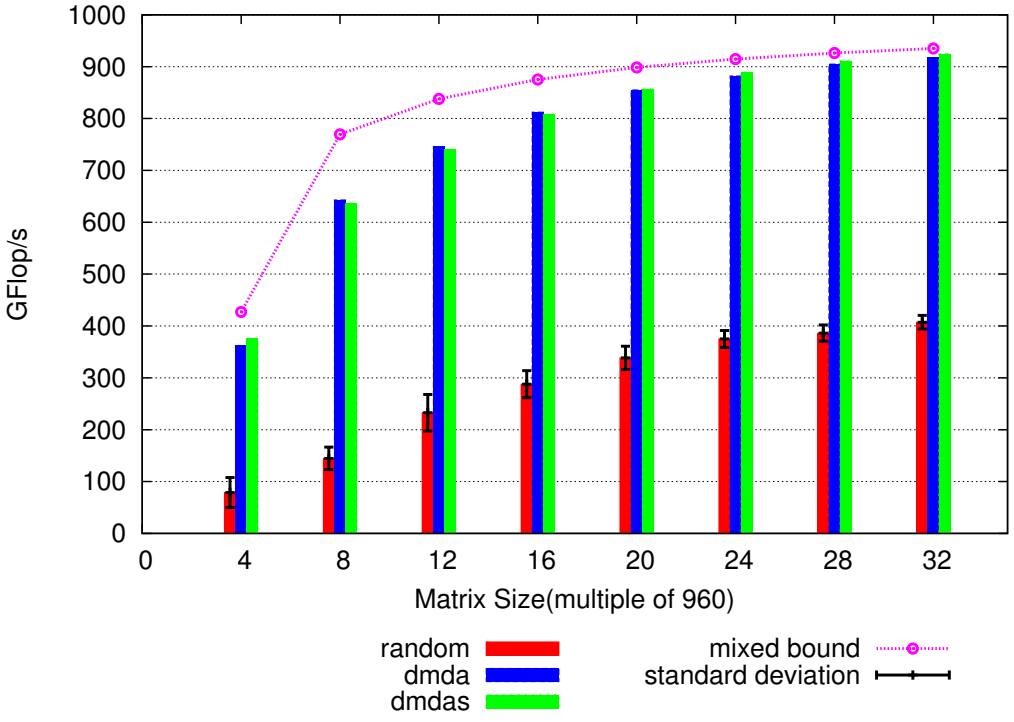


Figure 2.5: Heterogeneous related simulated performance.

Heterogeneous Related Case Figure 2.5 shows the simulated performance with different schedulers on the fictitious heterogeneous platform. Here, we can

observe 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.

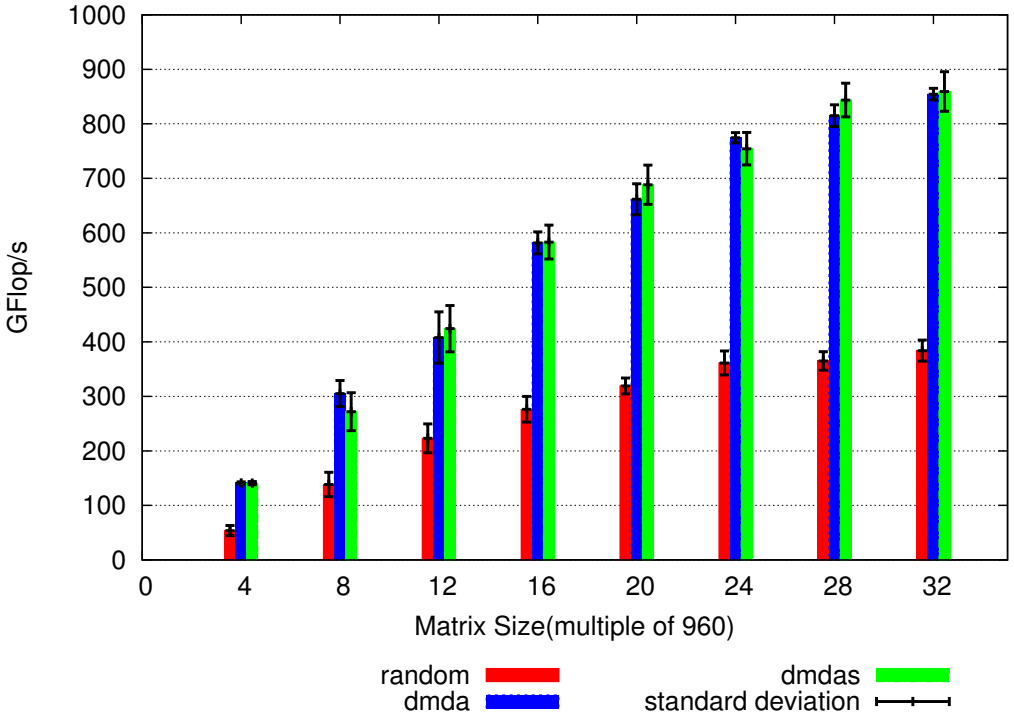


Figure 2.6: Heterogeneous unrelated actual execution performance.

As shown in Figure 2.6, 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 results in a lot of data movements 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 (selecting critical tasks versus tasks which generate high level of parallelism).

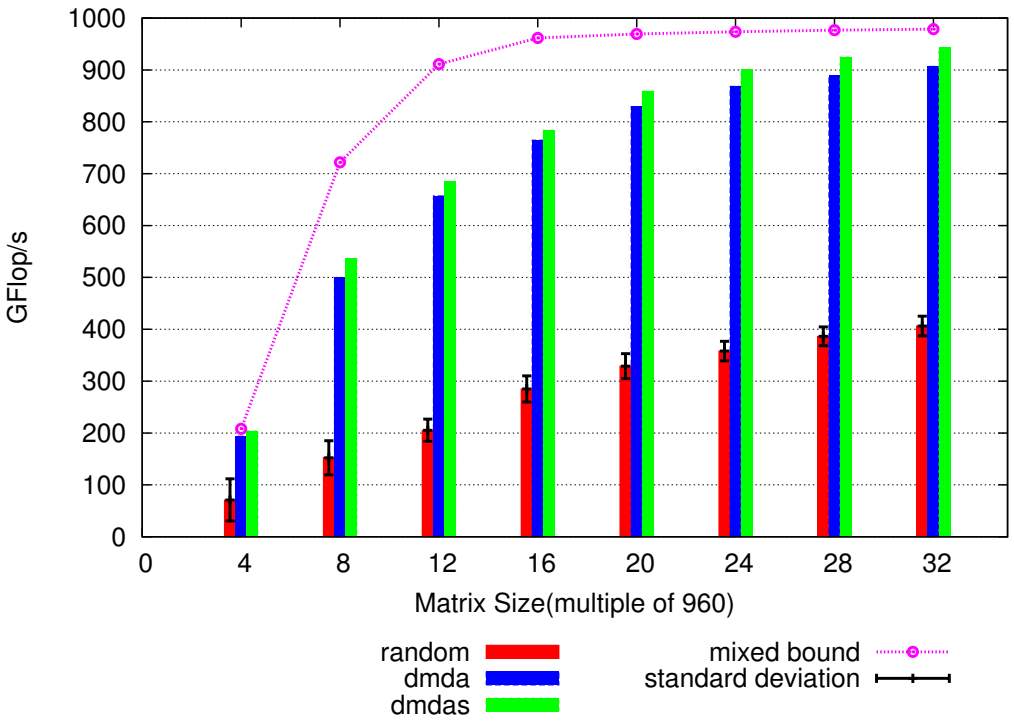


Figure 2.7: Heterogeneous unrelated simulated performance.

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 2.7 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 2.8, which can now be compared with the unrelated case of Figure 2.7.

Here we can see that unrelated speed-ups make the problem harder. That is

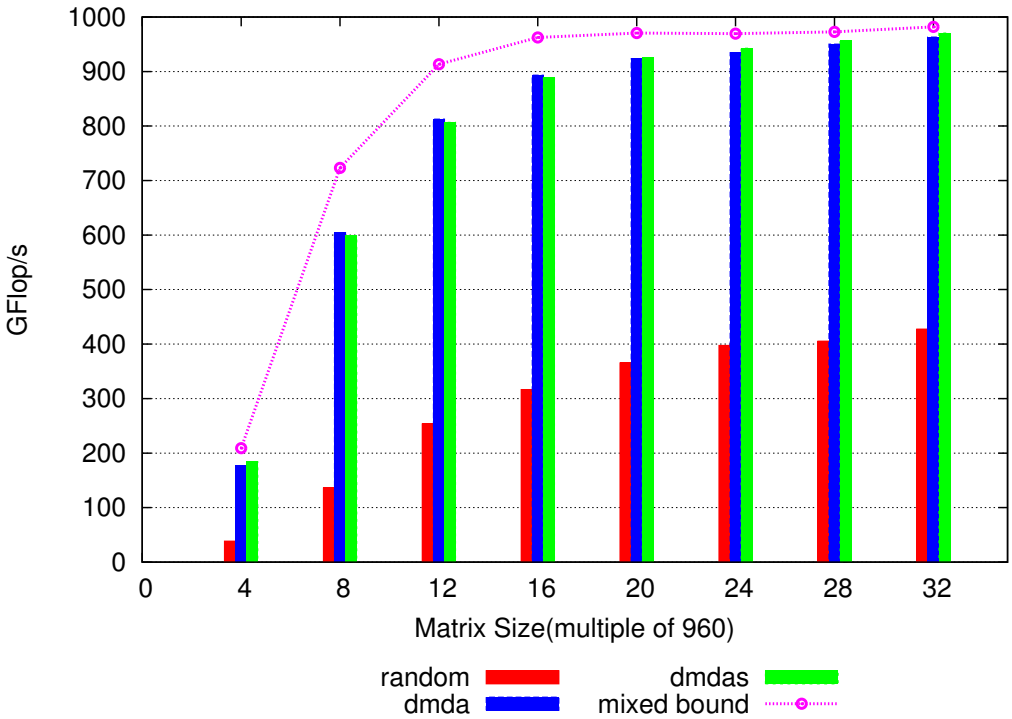


Figure 2.8: Heterogeneous related simulated *scaled performance*.

why the gap between state-of-the-art schedulers performance and *mixed bound* is large in Figure 2.7 compared to Figure 2.8. Here, it is also clear that there is room for improvement in the case of small and medium size matrices in the heterogeneous case.

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 2.7 highlights the following things:

- either the dynamic schedulers of StarPU return schedules 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 thus 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 2.9) and plotted the best obtained performance in Figure 2.10. 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 2.10 exhibits 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.

We eventually used the constraint programming (CP) described in Section 2.3.2 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 2.10) 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 2.10) compared to theoretical performance, 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

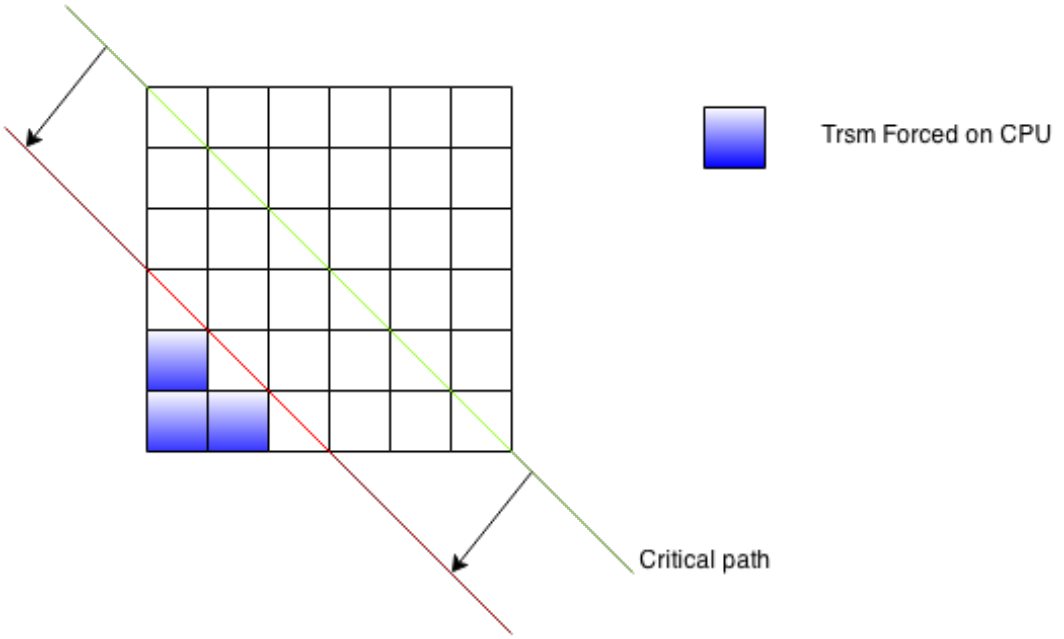


Figure 2.9: TRSMs forced on CPUs.

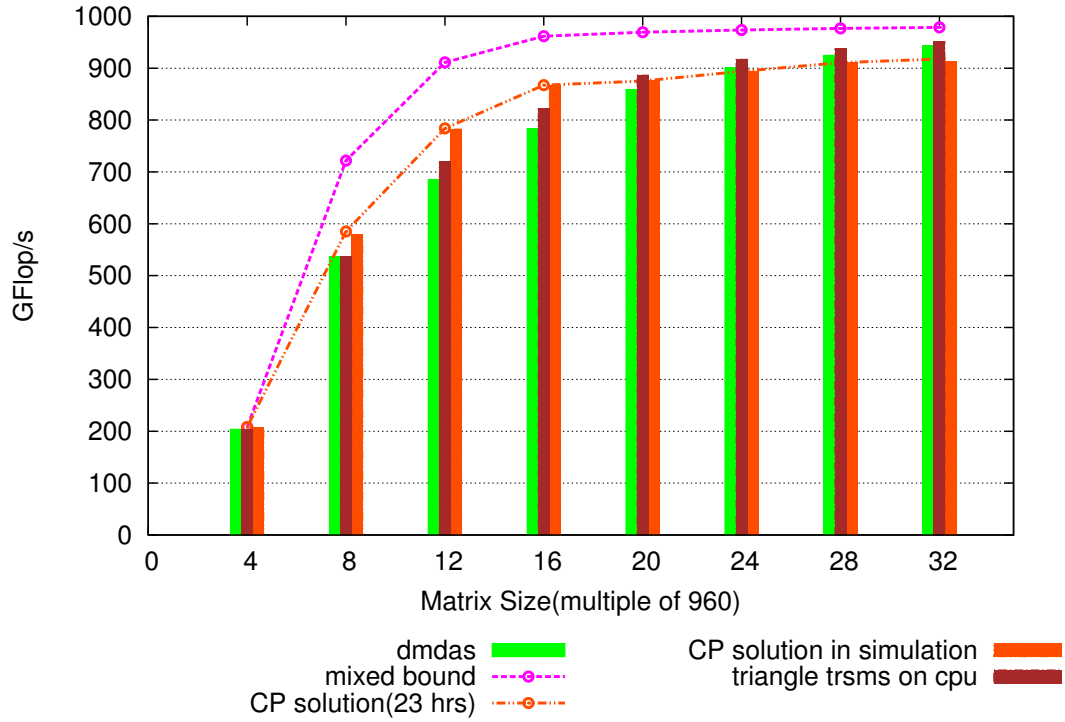


Figure 2.10: Heterogeneous unrelated simulated performance with static knowledge.

triangular information, force all TRSMs on CPUs which are at least k ($1 \leq k < \textit{Number of Tiles}$) tiles away from diagonal as static knowledge. Figure 2.11 shows the best performance in actual execution among obtained performance with all possible values of k.

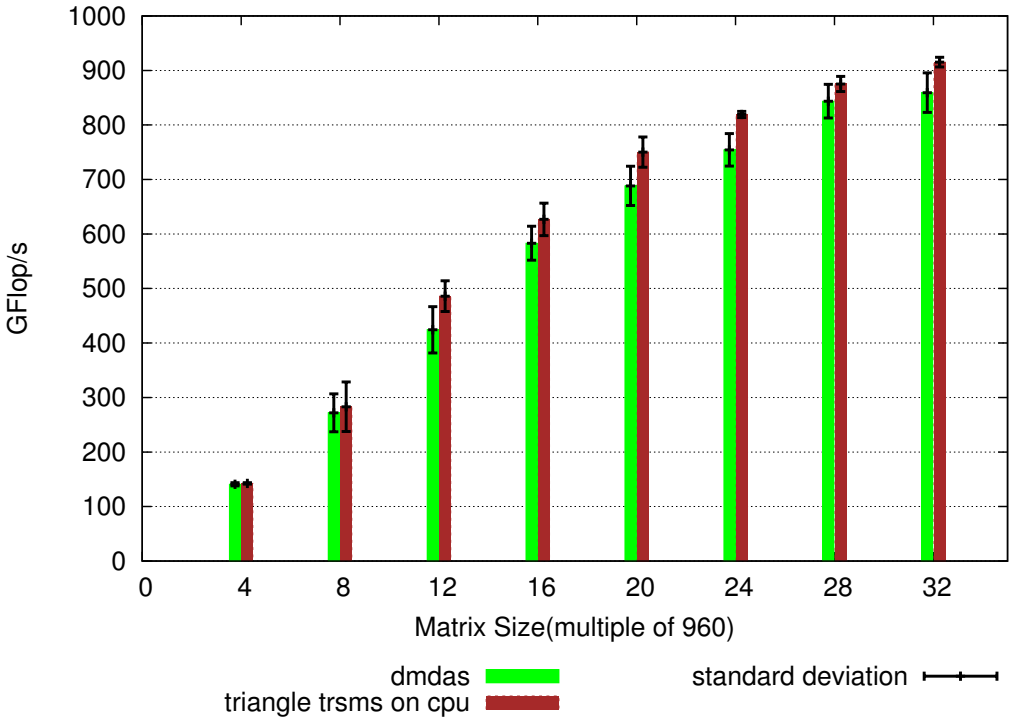
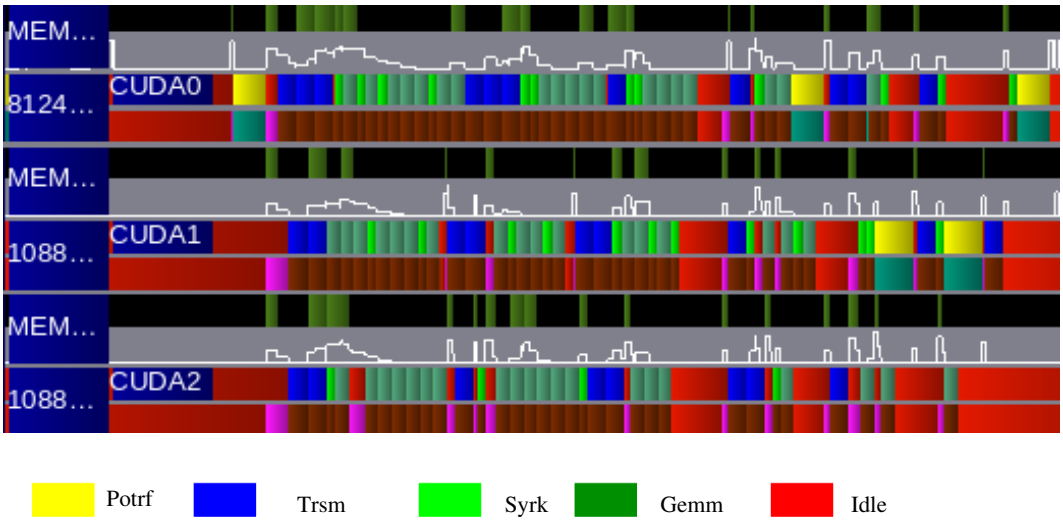


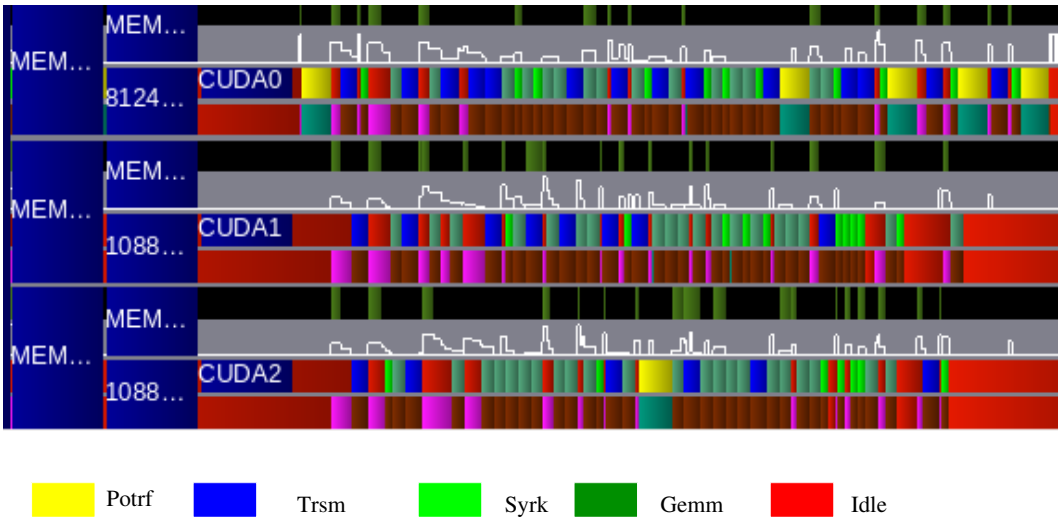
Figure 2.11: Heterogeneous actual execution performance with static knowledge.

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 2.3.2, solving a CP with data transfers has shown intractable for the purpose at stake. Actual execution with CP schedule thus adds a lot 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 thesis.

2.5 Discussion



(a) GPU trace with dmda scheduler.



(b) GPU trace with dmdas scheduler.

Figure 2.12: GPU Traces for 8×8 tiles.

2.5.1 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 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 2.12 shows traces with *dmda* and *dmdas* schedulers.

2.5.2 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.

2.5.3 Constraint Programming Schedule in Actual Execution

We did some experiments by injecting the schedule obtained by Constraint Programming in actual execution for smaller matrices, but the performance improvement was not significant compared to the state-of-the art schedulers. After looking into traces we found that resources are idle for significant portion of time during data transfers. One of the prominent ways to minimize the idle time on resources due to data transfers is to use prefetching in computational order, but using the prefetching very early also adds significant idle time on resources. Consider two data dependent tasks (second task is dependent on the data of first task) scheduled on two different workers with different memories. After execution of the first task, the second task becomes ready and will initiate the data transfer request. Due to serialization of transfers imposed by the GPU driver, it can be served only when all already initiated data transfers by second worker are completed (some of these data transfers may correspond to tasks which will be executed very late), which may keep the second worker idle for significant time. One of the heuristics to minimize idle time on resources is to use limited prefetching but even this strategy will not solve the problem completely. Since performance with CP schedule is highly dependent on task order of whole schedule, therefore adding idle time on one of the resources may

create idle time on other resources as well and degrades the overall performance dramatically.

2.6 Conclusion

In this chapter, 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.

More generally, this work opens a bridge to close 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. We also plan to study different static strategies with dynamic corrections, so that we can provide a fair comparison between static and dynamic scheduling strategies.

Chapter 3

Static vs Dynamic Scheduling Strategies

In previous chapter, we proposed different performance bounds for the scheduling of task graphs and analyzed the performance of dynamic schedulers based on actual executions and simulations. We also exhibited that adding static information of the application to the dynamic task schedulers improves the performance of the application significantly. In this chapter, we provide a deep analysis of Cholesky factorization on platforms consisting of GPUs and CPUs. Recall that this application encompasses many important characteristics in our context. It involves 4 different kernels (POTRF, TRSM, SYRK and GEMM) whose acceleration ratios on GPUs are strongly different (from 2.3 for POTRF to 29 for GEMM) and it consists in a phase where the number of available tasks is large, where the careful use of resources is critical, and in a phase with few tasks available, where the choice of the task to be executed is crucial. We analyze the performance of static and dynamic strategies and we propose a set of intermediate strategies, by adding more static (resp. dynamic) features into dynamic (resp. static) strategies. Our conclusions are somehow unexpected in the sense that we prove that static-based strategies are very efficient, even in a context where performance estimations are not very good.

3.1 Introduction

Our goal is to precisely assess the advantages and limitations of static (executed with possibly wrong estimations of execution times) and dynamic (computed online with basic greedy heuristics) strategies. We also design and evaluate a large set of intermediate solutions, by providing more static information to dynamic schedulers and by incorporating dynamic features into static schedules. Our study is rather deep than broad. In order to compare both approaches, we concentrate on a single dense linear algebra kernel, namely the Cholesky factorization (see description in Algorithm 2) on a single computing node con-

sisting of CPUs and GPUs, and we compare and analyze the results under a variety of problem sizes for a large set of sophisticated schedulers. To simplify the comparison of the different approaches, we assume that it is possible to overlap communications with computations, and we do not explicitly take communication costs into account.

The outline of the chapter is the following. Section 3.2 describes the tile Cholesky factorization algorithm and our experimental framework. In Section 3.3, we briefly describe works related to known static and dynamic schedulers for dense linear algebra kernels. We discuss different theoretical performance bounds and propose an improved bound, namely *iterative bound*, in Section 3.4. In Section 3.5, we discuss static strategies. In order to obtain the best possible schedule, we propose to use a constraint program (CP) whose use is limited to small and medium size problems due to its high cost. Then, we study the stability of optimal schedules under perturbations in kernel execution times. Using a large set of simulations, we prove that the optimal or close to optimal static schedule is in fact robust to realistic perturbations, and we furthermore add a dynamic work stealing strategy to better cope with those perturbations. In Section 3.6, we study the behavior of the dynamic schedulers that can be found typically in runtime systems such as StarPU. We prove that these runtime systems make poor use of slow (CPU) resources, restricting their use to POTRF kernels for which they are best suited. This is due to very conservative allocation strategies, that we alleviate using sophisticated prediction schemes in order to improve their efficiency. In Section 3.7, we introduce a new class of dynamic schedulers, that are easy to implement. We prove that it is possible to improve their efficiency when injecting simple qualitative knowledge about the application. Then, we compare the best variants of all three approaches in Section 3.8 and we prove that static based scheduling strategies are better than dynamic ones, even in presence of bad performance estimates, what is an unexpected result. In Section 3.9, we perform some experiments in actual execution based on information obtained from static schedules and we finally propose conclusions and perspectives in Section 3.10.

3.2 Context

3.2.1 Tile Cholesky Factorization

We recall the tile Cholesky factorization described in Chapter 1.4.1. Algorithm 2 for instance shows the pseudo-code of the tile version of the Cholesky factorization. This sequence of computation can be represented with a DAG (Directed Acyclic Graph) of tasks as depicted in Figure 3.1 in the case of 5×5 tile matrix. Throughout this chapter, the color code for the different kernels presented in Algorithm 2 and in Figure 3.1 will be used.

Algorithm 2: Tile Cholesky Factorization.

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

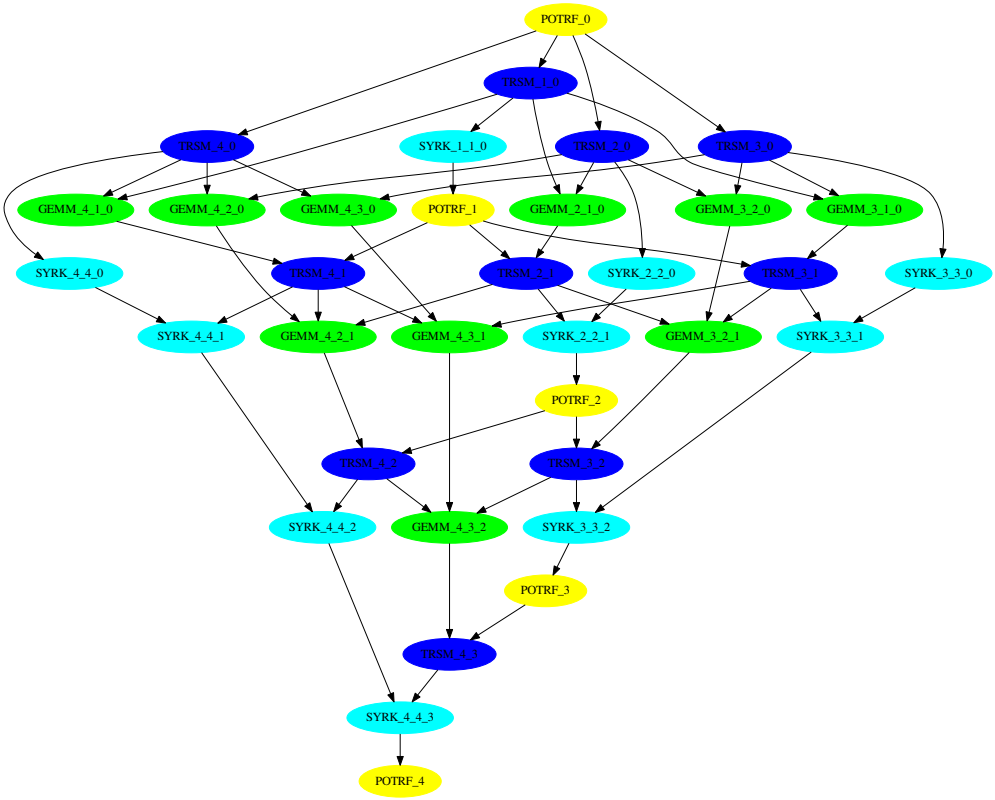


Figure 3.1: DAG of the tile Cholesky factorization - 5×5 tile matrix.

3.2.2 Experimental Framework

The heterogeneous platform and the library used to obtain the different Cholesky task timings in this chapter are the same as what we used in Chapter 2.

We consider a platform composed of nodes of two hexa-core Westmere Intel Xeon X5650 processors (12 CPU cores per node) and three Nvidia Tesla M2070

kernel	PORTF	TRSM	SYRK	GEMM
CPU time/GPU time	≃2.3	≃11	≃26	≃29

Table 3.1: GPU acceleration ratio over CPU core for all four kernels.

GPUs (3 GPUs per node). As most runtime systems, StarPU dedicates one CPU core to efficiently exploit each GPU. As a consequence, we can view a node as being composed of 9 CPU workers and 3 GPU workers.

Task timings of different Cholesky tasks have been obtained with the CHAMELEON [8] library running on top of the StarPU runtime system to assign tasks onto CPU cores or GPUs. CHAMELEON processes CPU tasks with the (sequential) Intel MKL library and GPU tasks with the MAGMA (POTRF kernel) or CUBLAS (other kernels) libraries. Consistently with [13], a tile size of 960 is being used. We observe that, for a given kernel and a given resource type, execution timings have relatively low variance: within the ±5 % of mean execution timing.

3.2.3 Comparing Static and Dynamic Schedulers

As stated in the introduction, our goal is to compare static and dynamic approaches when scheduling a DAG on a node consisting of both GPUs and CPUs. We recall that the Cholesky factorization is an excellent candidate to perform such a study. First, it is based (see Algorithm 2) on four different kernels that exhibit strongly heterogeneous performance and unrelated acceleration ratios on CPU cores and GPUs, as depicted in Table 3.1.

Second, despite its regular nature, the Cholesky factorization induces complex dependencies and leaves a lot of freedom for scheduling. Indeed, the i-th POTRF releases $N-i-1$ TRSMs and these TRSMs release $N-i-1$ independent SYRKs and $\frac{(N-i-2)(N-i-1)}{2}$ independent GEMMs (see Algorithm 2). Moreover, dependencies between the different kernels are not trivial and there is no need to synchronize all kernels involving i, the outer loop index. For instance, the execution of most of the GEMMs induced by i-th POTRF can be delayed and/or delegated to slow resources (GEMM_4_3_0 or SYRK_4_4_0 of Figure 3.1 can be delayed or delegated to slow resource).

Third, depending on the problem size, the underlying scheduling problems are of very different natures. Throughout this chapter, all problem sizes will be expressed in terms of number of blocks, the tile size being maintained constantly equal to 960 as mentioned earlier. Given the size of our platform, in the 8×8 case, it is crucial to perform tasks on the critical path as fast as possible, and it is not efficient to make use of all available resources. On the other hand, in the 32×32 case, the scheduling problem is almost amenable (except at the very beginning and at the end) to an independent tasks problem, and the crucial issue is to make use of all available resources in a proportion

that depends on the acceleration ratios given in Table 3.1. Intermediate cases, such as the 12×12 case, are typically hard, since both conflicting objectives (making an efficient use of resources and focus on the critical path) have to be simultaneously taken into account.

3.3 Related Work

The problem of scheduling tasks with dependencies has been highly studied in the literature, starting from complexity and approximation analysis from Graham et al. [64]. Many dynamic algorithms have been proposed to solve this problem, in particular for the homogeneous case. In the specific setting of Cholesky factorization, reversing the task graph allows to identify provably optimal schedules for the homogeneous case, and the problem is now well understood [38, 39, 83].

For the heterogeneous unrelated case, the literature is not as large. Most dynamic strategies are variants of the well-known heuristic HEFT [99] which combines a prioritization of tasks by their distance to the exit node with a greedy strategy which places tasks so as to finish as early as possible. Other noteworthy approaches are based on work stealing [33], where idle resources steal available tasks from other resources, or on successively applying an algorithm for independent tasks scheduling on the set of ready tasks [31]. More static approaches have also been proposed to obtain more efficient schedules at the cost of longer running times. For instance, Constraint Programming is a paradigm which is widely used to solve many scheduling problems [24]. Branch-and-bound algorithms can also be designed for scheduling problems, with a wide range of search strategies [94], but the weakness of bounds in the heterogeneous case makes them less efficient than in the homogeneous case.

3.4 Iterative Bound

In this chapter, we also use upper bounds on performance (lower bounds on execution time) to assess the quality of the obtained schedules. Classical bounds in the homogeneous case are the area bound, defined as the total work divided by the number of processors, and the critical path, which is the maximum execution time over all paths in the graph. For the heterogeneous case, the area bound needs to be adapted, and can be defined as the solution of a linear program which expresses how many tasks of each type are scheduled on each resource. The critical path can also be expressed, however better results can be achieved when computing both bounds simultaneously, since this allows to express the tradeoff for critical tasks: if they are executed on faster resources but with poor acceleration, they improve the critical path but degrade the

area bound. Such a mixed bound has been described in Chapter 2, and in this chapter we use an improved version, namely *iterative bound*.

We know that each task i has to be executed on the whole platform, which has M_r number of processing elements of type r. We focus on i_r, which denotes the fraction of task i processed on resource type r. For each task i and resource type r, we also know the execution time T_{ri} (obtained by the calibration mechanisms of StarPU). The *iterative bound* is obtained by solving the following linear program.

minimize the makespan l such that

$\forall i$, each task i get executed:

$$\sum_r i_r = 1$$

$\forall r$, the M_r resources of type r complete all their tasks of various types t within the makespan l:

$$\sum_r i_r T_{ri} \leq l \times M_r$$

length of each path $a \to b \to \cdots \to c$ is within makespan:

$$\sum_r a_r T_{ra} + \sum_r b_r T_{rb} + \cdots + \sum_r c_r T_{rc} \leq l$$

$\forall r$, $i_r \in [0, 1]$

While implementing the above linear program, we formulate the path constraint in a different way. We first obtain the solution of linear program with the first two constraints and look for a path in the task graph that is longer than computed makespan. If such a path is found we also add length of this path as a constraint and iterate the whole procedure again. This formulation guarantees that all path lengths of a task graph are within computed makespan l. We use this formulation in this chapter to compute iterative bounds for different task graphs.

Another way to express path constraint is to add constraints for each dependency and end time of each task. Since we take all dependencies into account, therefore this formulation ensures that each path length is within makespan l. But in this case we have to consider one extra variable to represent start time s_i for each task i. This formulation requires to solve linear program only once while the previous formulation solves linear program in each iteration. We use this formulation in later chapters to compute iterative bounds for task graphs. The complete linear program to compute *iterative bound* with this formulation is the following:

minimize the makespan l such that

$\forall i$, each task i get executed:

$$\sum_r i_r = 1$$

$\forall r$, the M_r resources of type r complete all their tasks of various types i within the makespan l:

$$\sum_r i_r T_{ri} \leq l \times M_r$$

$\forall i$, task i completes within makespan l:

$$s_i + \sum_r i_r T_{ri} \leq l$$

$\forall i \to j$, dependency $i \to j$ is respected:

$$s_i + \sum_r i_r T_{ri} \leq s_j$$

$\forall r$, $i_r \in [0, 1]$

3.5 Static Strategies

In this section, we describe schedules obtained with a Constraint Programming formulation for the scheduling problem proposed in Chapter 2, and we analyze their robustness to errors in computation times. The computing time needed to obtain a good schedule depends on the size of the task graph (number of tasks and dependencies) and of the platform description (number of choices for each task). In our case, the number of choices is limited to deciding whether a task is allocated to CPU or GPU; however the number of tasks grows as a cubic function of the matrix size. For this reason, it is possible to obtain nearly optimal solution for small matrices and good solution for intermediate matrices in a few hours. But the solutions obtained for large matrices are far from optimal, and most of the dynamic strategies achieve better timings than those solutions (see dynamic strategies). Figure 3.2 provides a comparison of the solution obtained from this formulation with the bounds discussed in the previous section. This graph also shows how the iterative bound is able to improve over previous bounds, and how the CP formulation is able to compute almost optimal solutions for small cases.

In order to determine the stability of CP schedules, we use 30 different sets of execution timings by introducing some randomness ($\pm 10\%$) in the original execution timings of tasks on each resource. We normalized the execution timings with respect to the *area* bound of the corresponding task graph, so that

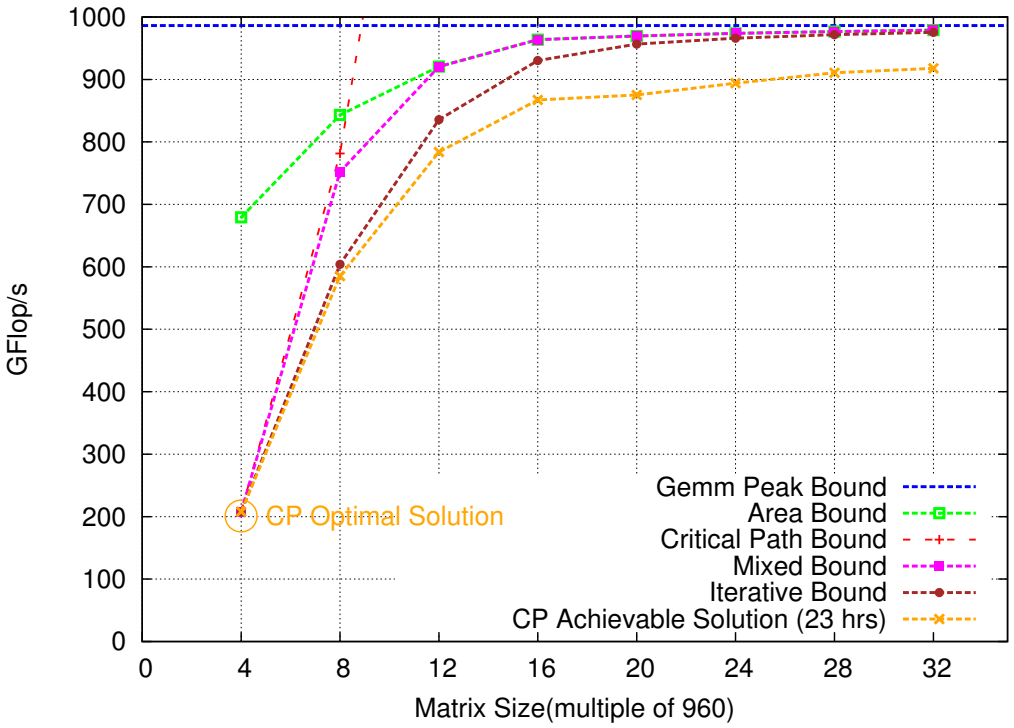


Figure 3.2: Upper bounds and CP feasible solution performance.

the *area* bound of all sets of execution timings for a given matrix size corresponds to the same value. For each of these generated execution timings, we use the same static schedule (obtained with CP formulation using the original timings) by keeping on each resource the same allotted tasks in the same order (of course start times may be different because of the changes in execution times of tasks). Figure 3.3 shows the performance ratio of each of the obtained schedules compare to the *iterative bound* for the 12×12 tile matrix, in which experiment number 0 corresponds to the original execution timings. On other experiments, the performance degradation is below 10 % compared to the performance ratio with the static schedule on the original timings. Using this static schedule can therefore be a reasonable option for intermediate size matrices – but obtaining a good solution is the hard part. In all the rest of the chapter, the static strategy will be denoted as SS.

3.5.1 Some Dynamic Strategies with Static Schedule

Though performance degradation is limited in presence of perturbations, we observed in the obtained schedules that some GPU resource remain significantly idle in some experiments. Figure 3.4a shows the trace of one of the experiments with perturbed execution timings. Here one of our critical resource (GPU1) is idle for a significant amount of time in the middle of the

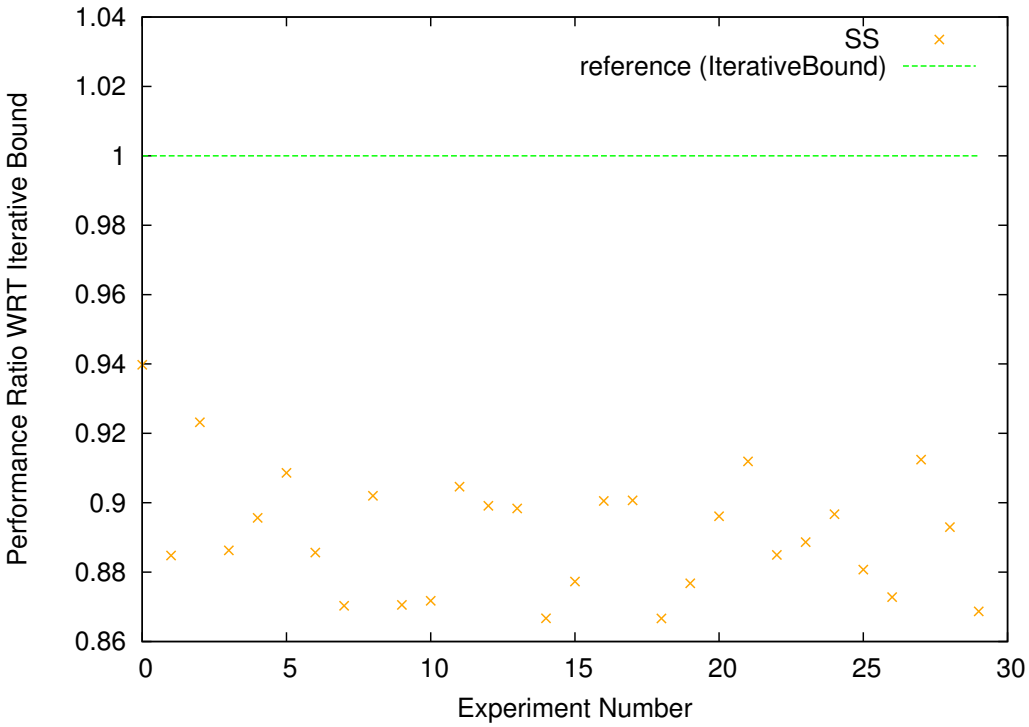
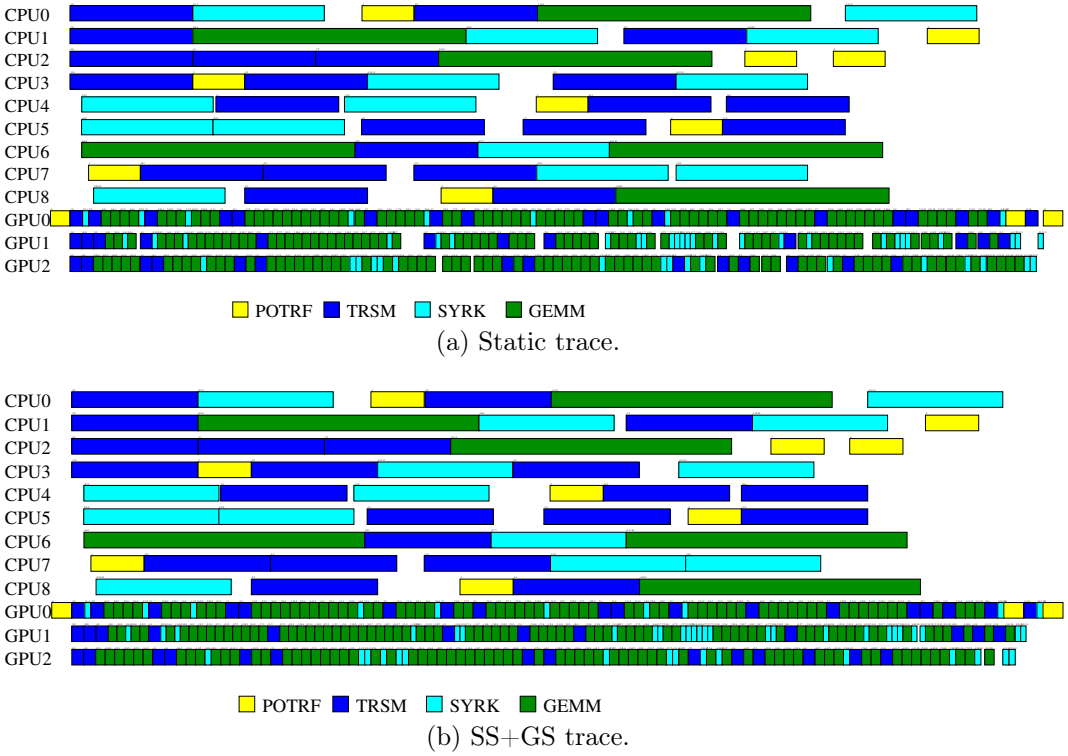


Figure 3.3: Performance ratio of static schedules with respect to *iterative bound* - 12 × 12 tile matrix.

execution, because the next task that should be executed on this resource is not ready yet (remember that we keep the order as given by the CP solution). This observation is a motivation to improve the performance by injecting dynamic corrections to the static schedules.

The acceleration factor of GEMM tasks is highest on GPU among all Cholesky tasks. Therefore we allowed an idle GPU worker to help other workers by executing the GEMM tasks of other workers. When a GPU worker is idle and waiting for some task to become ready, it searches for the highest priority ready GEMM in its own list and then in the list of other workers, and executes it if one is found. We name SS+G such a correction to the original SS. This strategy improves the performance of SS slightly but does not eliminate all idle time from GPUs in the middle of execution. Therefore we also consider stealing SYRK tasks, whose acceleration factor on GPU is second highest (after GEMM acceleration factor). We name SS+GS such a correction to the original SS. Figure 3.4 shows the comparison of trace with SS (Figure 3.4a) and SS+GS (Figure 3.4b).

Figure 3.5 shows that allowing some dynamic strategies with SS improves the scheduler performance in most of the experiments. while its adverse effect on a few experiments is very negligible (performance degradation is less than 1 %). This allows to obtain good and stable solutions when compared to the



(a) Static trace.

(b) SS+GS trace.

Figure 3.4: Trace with perturbed timings for 12×12.

iterative bound, even in presence of noisy execution timings.

3.6 Heft-like Solutions (Dynamic, Task-centric)

We use *heft* (heterogeneous early finish time) and *heftp* (heterogeneous early finish time with priority) schedulers, which are based on a very well know state-of-the-art task centric *HEFT* heuristic. When a tasks is ready, both algorithms put it in the queue of the resource that is expected to complete it first, given the expected available time of the resource and the expected running time of the task on this resource. The only difference between *heft* and *heftp* is the use of priorities. In *heftp*, task priorities are computed offline based on the longest path from the task to last task in the DAG, using minimum expected execution timing of each task in presence of heterogeneous resources, as proposed in [99]. Then, *heftp* schedules ready tasks ordered by their priorities to worker queues and in turn, each worker selects and schedules the task from its own queue with the highest priority.

Note that these schedulers are similar to *dmda* and *dmdas* scheduler of StarPU. We are using our own simulator, which is why we have not used StarPU specific scheduler names.

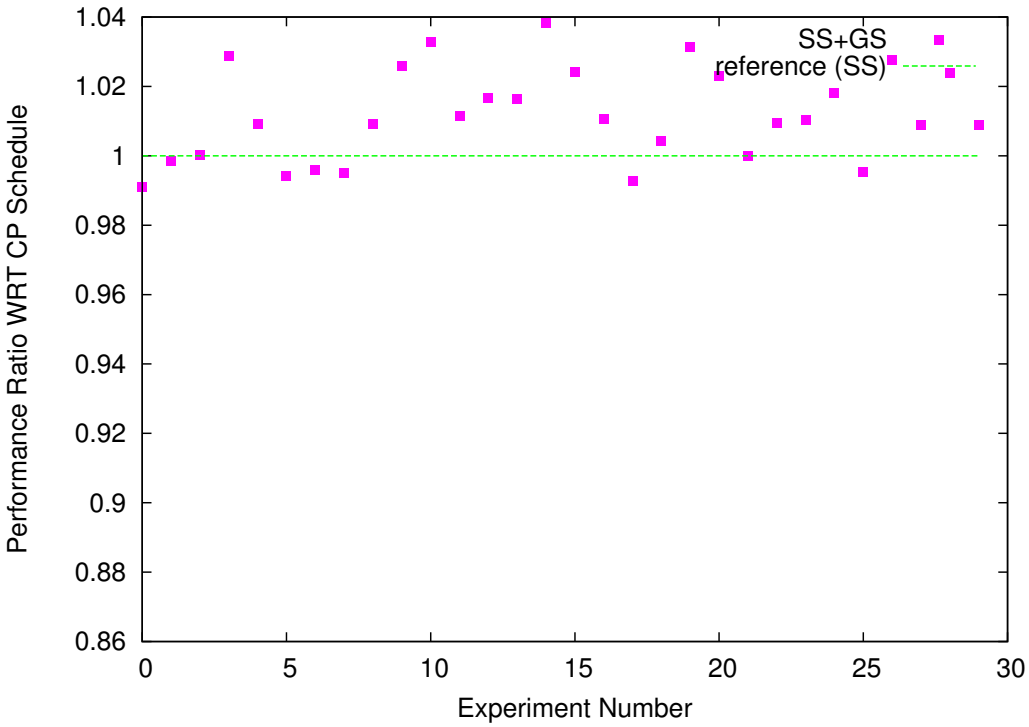


Figure 3.5: Performance improvement of SS+GS over original SS obtained with CP - 12 × 12 tile matrix.

In Figure 3.6, we can observe that *heftp* outperforms *heft* for all matrix sizes, thanks to its capability of executing tasks close to the critical path.

Although *heftp* outperforms *heft*, we can observe on Figure 3.7 that the allocation dynamically computed by *heftp* is far from optimal since it wastes most of CPU resources. Indeed, only CPU0 is used to process tasks, and its use is restricted to the execution of POTRFs, that are the most efficient kernel on CPUs (see Table 3.1). Therefore, in practice, *heftp* is too conservative and the running times on CPUs and GPUs are so different that for all tasks (excepts a few POTRFs), the expected completion time is always smaller on one of the GPUs. On the other hand, we have observed that there are tasks (typically GEMMs and SYRKs) that are released early in the execution but whose results are needed late. Such tasks are typically good candidates to be executed on CPUs. In practice, they are released early and at the time when they are allocated by *heftp* on a GPU, their expected completion time is small. On the other hand, since their priorities are low, they will be consistently passed over by other tasks in the GPU queue and their actual completion time on the GPU is in fact larger than their expected completion time on a CPU.

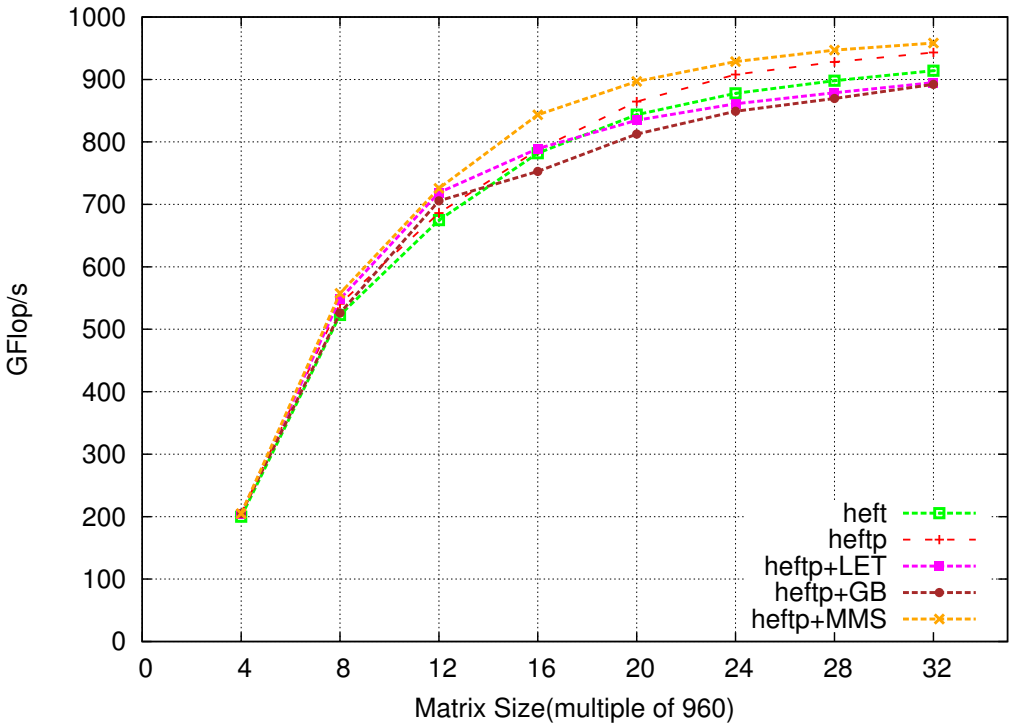


Figure 3.6: Performance with different heft schedulers.

3.6.1 Improvement of *heftp* Scheduler

Following this observation and in order to improve *heftp* performance by making good use of all CPU resources , we modify *heftp* scheduler such that scheduling decision is not only based on the minimum completion time heuristic but also based on certain look-ahead information.

heftp+LET (Local Execution time)

In this strategy, before making a scheduling decision for a ready task t, the scheduler first computes the minimum expected completion time on a CPU

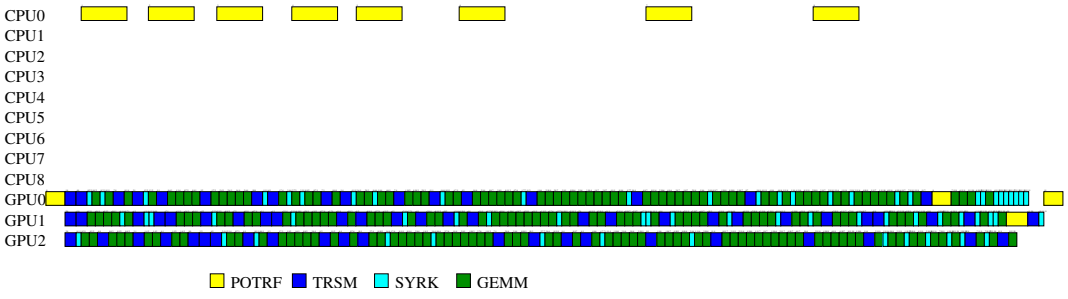


Figure 3.7: *heftp* trace for 12 × 12.

(e_{cpu}) and then **simulates** the execution of *heftp* until task t completes execution (e_{heftp}) on some (GPU) worker. If $e_{cpu} \leq e_{heftp}$, that typically corresponds to the situation described above where t has been passed over by many higher priority tasks in the GPU queue, then *heftp+LET* schedules task t on a CPU.

heftp+GB (GPU Busy)

In this strategy, *heftp+GB* always tries to assign task t on a CPU and then **simulates** the execution of *heftp* until the completion time of task t on a CPU (e_{cpu}). Then, it checks whether all GPUs have been busy between the current time and e_{cpu}. If it is the case, then *heftp+GB* assumes that it is safe to schedule t on a CPU; otherwise, t is scheduled on a GPU.

heftp+MMS (Min Makespan)

In this (higher cost) strategy, scheduler selects for task t a CPU worker if and only if it improves the overall makespan. To determine whether it is the case, *heftp+MMS* **simulates** the execution of *heftp* until the end, with t forced on a CPU and t allocated according to *heftp*. If the simulation time is smaller with t forced on a CPU, then *heftp+MMS* allocates t on this CPU.

3.6.2 Analysis of Different Improved *heftp* Schedulers

Figure 3.6 describes the performance of the different *heftp* heuristics. *heftp+LET* and *heftp+GB* strategies use simulation upto a certain lookahead (until task completes its execution in heftp+LET and until task completes its execution on CPU in heftp+GB). Therefore, this adds an acceptable overhead to the scheduler and it results in a larger use of CPU resources. This induces a positive effect on the overall makespan for small to medium size cases, as shown in Figure 3.6.

But making the good utilisation of CPUs does not guarantee to improve the performance! Indeed, *heftp+LET* and *heftp+GB* strategies schedule a significant amount of GEMMs and SYRKs on CPUs, *i.e.* tasks that are not well suited to CPUs (see Table 3.1). On the other hand, when the size of the problem becomes large, the problem is of different nature, since all heuristics keep all resources (CPU and GPU) busy most of the time. Then, the critical path bound becomes less important than the area bound and what becomes crucial is to allocate tasks on the best suited resources, what is better achieved by *heftp*. *heftp+MMS* strategy is based on the estimation of the overall completion time, and therefore does not suffer from these limitations for large sizes (see Figure 3.6). On the other hand, it induces a (too) large scheduling overhead to be used in practice, due to the simulation cost.

3.7 HeteroPrio-like Solutions (Dynamic, Resource-centric)

3.7.1 Baseline HeteroPrio Scheduler

HEFT-like heuristics are task-centric as they first select a particular task before attributing it to a particular resource. One drawback of this class of greedy heuristics is that they may attribute a considered task (say a POTRF) to a given resource (say a GPU) because at decision time it is the best suited with respect to the expected completion time, conducting not to schedule another available task (say a GEMM) to be executed on that resource whereas it would have been a better fit with respect to the acceleration factor. One option to overcome this limit consists of injecting static knowledge to the heuristic as discussed above. A more drastic alternative consists of designing another class of heuristics, resource-centric, that aim at selecting the task that achieves the best acceleration factor for a given resource. Such an approach is relatively natural in the case of independent tasks and was first introduced in [16] under the name of HeteroPrio (HP) to enhance task-based fast multipole methods (FMM) whose computation is dominated by independent tasks. We investigate such an alternative approach that we implement with the following design. Multiple scheduling queues are instantiated, each queue aiming at collecting tasks of acceleration factors of same magnitude. In the baseline version that we propose we consider one queue per type of task (hence four in total). Whenever a worker is idle it polls for a task within the set of ready queues and selects the one which best suits to this worker. In our case, CPU cores hence poll POTRF, TRSM, SYRK and GEMM queues whereas GPUs poll the queues in reverse order. To favor progress, within a queue, a GPU (resp. CPU) selects the highest (resp. lowest) priority task.

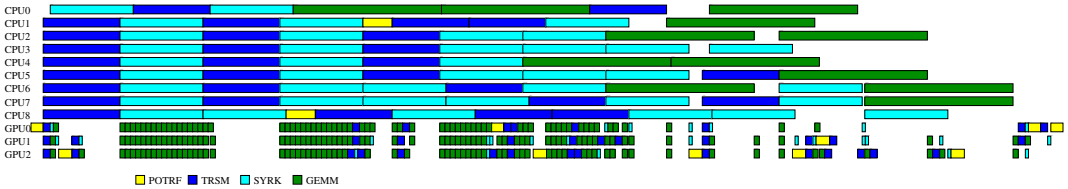


Figure 3.8: 12 × 12 trace with heteroprio scheduler.

Figure 3.8 presents a 12 × 12 HP execution trace. Because nothing prevents CPUs to process tasks when idle in the baseline version of HP, CPUs get attributed tasks that induce starvation on some GPUs while being executed, which may potentially lead to significant performance degradations. Furthermore, in this baseline version, progress is only ensured with the ordering of tasks within a queue. This strategy, which aggressively favors the acceleration of tasks, may be insufficient to ensure a global progress along the critical path,

eventually leading to starvation. This is why GPUs are periodically starving in the trace.

3.7.2 Improved HeteroPrio Algorithms

We now propose successive corrections (each correction contains previous ones) to the baseline version of HP in order to find a better trade-off between acceleration of tasks and progress.

HP+Sp

The first correction we introduce consists of preventing immediate GPU starvation thanks to the following spoliation (Sp) rule. When a GPU is starving while at least one CPU is executing a task, the execution of the highest priority task being executed on CPU is aborted and attributed to the GPU, provided it finishes the task earlier.

HP+CGV

Defining multiple queues for tasks whose acceleration factors are of roughly the same magnitude may provide only a limited advantage in terms of acceleration but a severe penalty in terms of progress. For this reason, in addition to HP+Sp correction, we propose that GPUs get a combined view (CGV) of GEMM, SYRK and TRSM ready queues whose acceleration factors are in a relatively thin range of values ($[11; 29]$) with respect to the distance to POTRF acceleration factor (2.3).

HP+PP

POTRF tasks have a very low acceleration factor with respect to other kernels, we propose to favor its execution on CPU with the following preemption rule in addition to HP+CGV correction. If all workers are busy when a POTRF becomes ready, the lowest priority task being executed on CPUs is aborted and set back to the ready queue so that the considered POTRF task can be immediately attributed to that CPU. In this case, preemption is only applied to POTRF, so we call it POTRF preemption (PP).

HP+PC

When a CPU is selecting a task, that task may have a relatively low priority with respect to other ready tasks in that queue. However other tasks with lower priority may become ready while that task is being processed. If a GPU becomes free at that time, it may thus have to pick up one of those new low priority ready tasks and potentially prevent fast progress on the critical path.

To overcome this issue due to the greedy nature of HP, we propose to forbid a GPU to pick up a ready task with a lower priority than a task being executed on CPUs. For that, we introduce the following additional spoliation rule. If no ready task has priority higher than all tasks being executed on CPUs, then the GPU spoliates the highest priority task being executed on CPUs. This additional spoliation enhances progress thanks to a Priority Constraint (PC). This strategy is quite restrictive and allows only few tasks to run on CPUs.

HP+PCEP

We therefore propose a variant where the previous PC spoliation rule does not apply to POTRF, which we name Priority Constraint Except POTRF (PCEP).

HP+PCEPT

If PC spoliation is excepted for both POTRF and TRSM, the rule is then called Priority Constraint Except POTRF and TRSM (PCEPT).

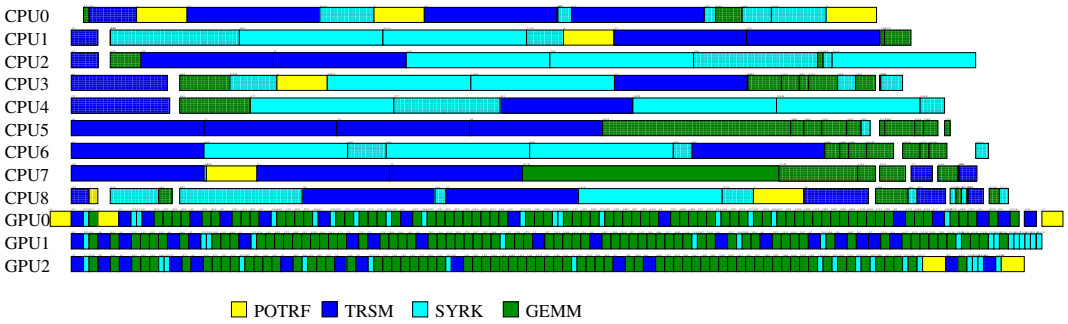


Figure 3.9: 12 × 12 trace obtained with *HP+PCEPT*. Aborted tasks on CPUs due to spoliation or preemption are represented in non solid boxes.

Figure 3.9 shows the resulting HP+PCEPT 12 ×12 execution trace, which achieves the best performance among all HP proposed variants for that matrix order. The proposed heuristic managed to schedule most POTRF tasks on CPU, while achieving a very high occupancy with well suited tasks on both GPUs and CPUs.

3.7.3 Performance Comparison of Heteroprio Variants

Figure 3.10 shows the performance of most relevant HP variants proposed above. Large matrices have relatively more number of independent tasks at different execution points, which is well suited to HP variants. That is why even baseline HP starts performing better as matrix size increases. HP+Sp performance indicates that spoliation rule is very useful when there are not

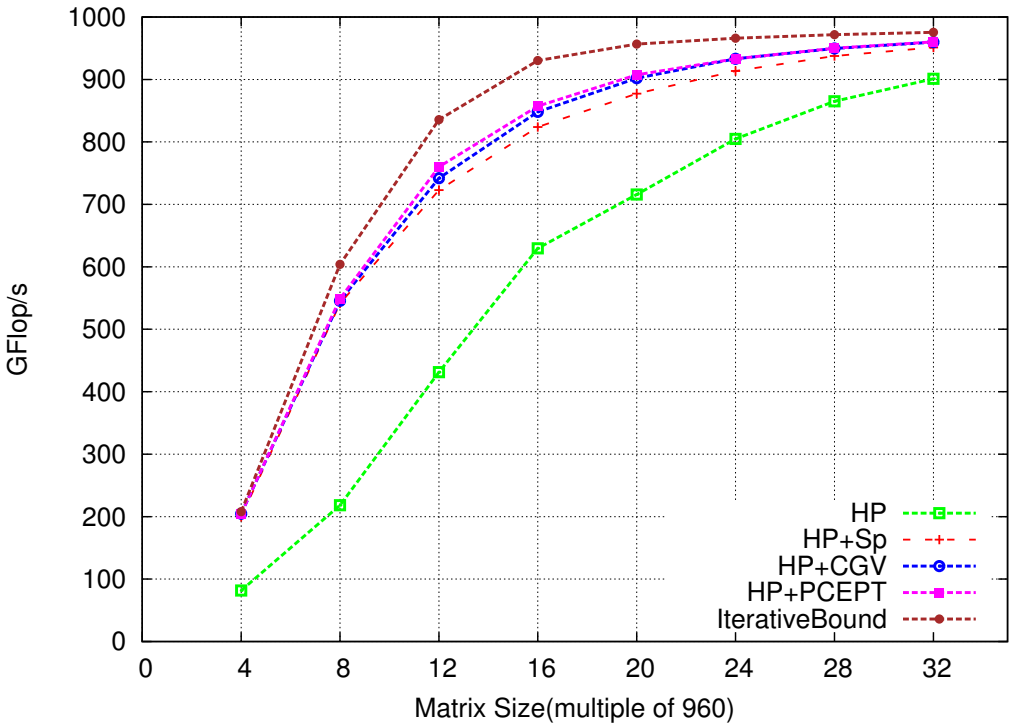


Figure 3.10: Performance with different HP schedulers.

enough independent tasks. Considering priority constraints with some relaxation (HP+PCEPT) improves performance for intermediate matrices and its performance is very close to the iterative bound for large matrices, which indicates that HP+PCEPT manages priorities (critical tasks) and tasks heterogeneity in well manner.

3.7.4 Feasibility of the Implementation of HP Corrections

The first implementation of HP proposed in [16] was implemented on top of StarPU following a twofold approach. The baseline version of HP was applied when enough ready tasks were available (HP was said to be in a steady state). When fewer tasks got in the system (HP was said to be in a critical state), CPUs were prevented to execute long tasks in order to ensure a fine termination. The corrections proposed in the present study are much more advanced and we discuss here the feasibility of the implementation. These corrections rely on three ingredients: combining queues, performing spoliation and preemption. Modern runtime systems such as StarPU provide infrastructure for designing user-level scheduling algorithms. In particular, dealing with user-level queues is natural and combining their GPU view immediate. On the

other hand, spoliation and preemption require to abort tasks and recover data of aborted tasks, which are not supported in most state-of-the art runtime systems. Most thread libraries provide functions to cancel or restart a thread execution, this can be used to abort a task. Data recovery of an aborted task might be technically difficult. However recent contributions have been proposed by runtime community to perform forward recovery in the context of resilience [71]. This mechanism could be applied to recover data of an aborted task and thus implement spoliation or preemption. Alternatively, speculative scheduling using simulation at runtime could be employed [96].

3.8 Comparison of All Three Approaches

In this Section, we propose to compare these three approaches (static, heftp, and HeteroPrio) in different cases: first with original execution timings as measured on the actual platform, then with perturbed timings which are constant throughout the execution (like in Section 3.5), and finally with perturbed timings within an execution.

3.8.1 Original Timings

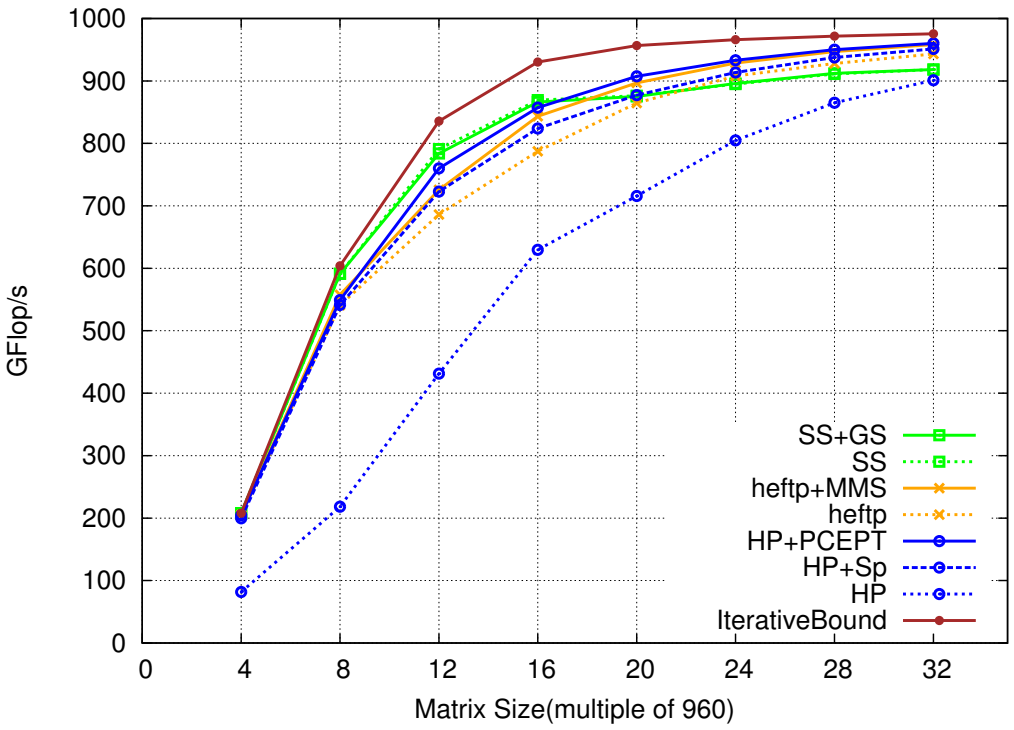


Figure 3.11: Performance with different types of schedulers.

Figure 3.11 shows a comparison between the best variants of different schedulers. The static schedule is obtained with these exact timings, which is why allowing movement of GEMM and SYRK tasks (SS+GS strategy) reduces the performance slightly in this case. On large matrices, computing a quality static schedule is very costly, and the CP formulation is only able to provide a low performance solution. For dynamic strategies, *HP+PCEPT* obtains consistently better performance than the best *heftp* variant (which is heftp+MMS), and both outperform the static schedule and obtain performance very close to the upper bound for large matrix sizes. On intermediate matrix sizes (12 or 16), all solutions are relatively farther from the bound, which may indicate that it would be possible to design stronger bounds.

3.8.2 Perturbed Timings

As indicated in Section 3.5, we consider 30 different sets of execution timings for each type of task on each resource, obtained by changing the original execution timings by ±10%. For consistency, these timings are then normalized to obtain the same *area* of the task graph as with original timings: all sets of execution timings for a particular matrix size will yield the same area bound. Unlike the previous case we provide here results about all variants discussed in this chapter. Figure 3.12 shows the distribution of the performance of each algorithm for all matrix sizes, where plots are grouped by matrix sizes. For each matrix size and each algorithm, the box on the plot displays the median, first and last quartile, and the whiskers indicate minimum and maximum values, with outliers being shown as black dots.

Figure 3.12 shows that the performance of all HP variants increases with matrix size. It follows from the fact that HP variants are very good with a large number of independent heterogeneous tasks. The performance of *heftp+LET* and *heftp+GB* degrades for large matrices due to their tendency to use the CPU resource greedily and thus allocate too many tasks which are well suited on GPU resource, as mentioned in Section 3.6.2. As previously, the static solution is not very good for large matrices and most of the dynamic schedulers have better performance in these cases. However, we can also observe the benefits of dynamic modifications of this static solution, which allow to cope with perturbation of timings. As discussed in Section 3.7.2, the restrictive nature of the HP+PC scheduler yields a poor performance compared to other HP+Sp variants. On the other hand, its relaxed version HP+PCEPT achieves the best performance among all dynamic schedulers for intermediate and large matrices.

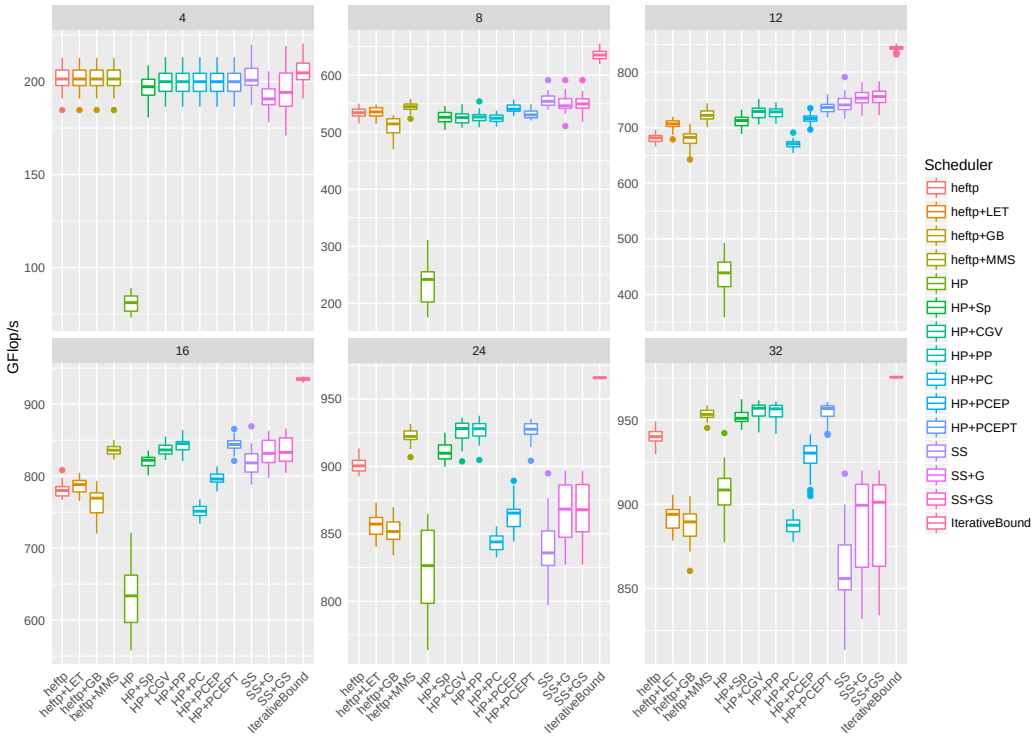


Figure 3.12: Comparison of different schedulers with perturbed execution timings. Title field of each subplot indicates corresponding matrix size.

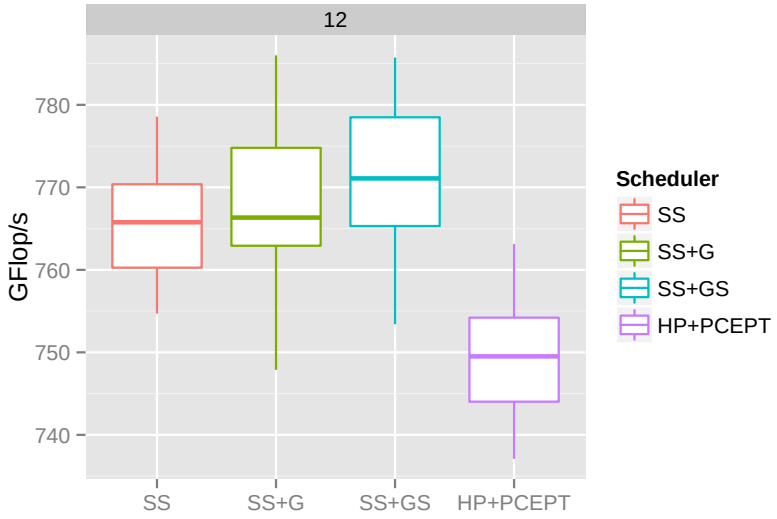


Figure 3.13: Comparison of all SS-based strategies with best HP variant - 12 × 12 tile matrix.

3.8.3 Perturbed Timings within an Execution

We now present the set of experiments, in which execution timings for a particular task on a particular resource is not constant. Each time a task is executed, its execution timing is randomly drawn between ±10% of the original execution time. We ran our experiments with 30 different random seeds and show in Figure 3.13 the performance of the best dynamic strategy and dynamic variants of static solutions for 12×12 matrix. This plot shows a behavior similar to other experiments for 12×12. It shows that even in this context, schedules based on the static solution always perform better than our best dynamic HeteroPrio scheduler (HP+PCEPT). It also shows that adding dynamic corrections to the static schedule tends to improve the overall performance in presence of perturbed timings.

3.9 Static Schedule in Actual Execution

In this section, we show how schedules obtained with previously explained scheduling strategies can be used in actual execution. We focus on 12×12 tile matrix and perform actual executions in different settings.

We did not consider communication costs while obtaining schedules with Constraint Program (in Section 3.5) or with dynamic strategies (in Sections 3.6 and 3.7). Therefore, we provide only partition information (CPU or GPU) of tasks from the schedule to the StarPU and let StarPU select exact worker based on MCT heuristic [103].

Figure 3.14 shows actual execution trace for 12×12 tile matrix, where partition information is provided based on the schedule obtained with HP+PCEPT strategy. It exhibits that most GPUs are significantly idle and data transfers among different memory units are not completely overlapped. To minimize idle time on GPUs due to non-overlapping of data transfers, we propose to calculate schedule where execution timings of all tasks on CPU are increased by some percentage. This strategy compensates for communication delays and allows GPUs to process other tasks in that time.

We experimented with 2, 5, 8, 10, 12 and 15 percent increase in CPU execution timings of tasks. We found that schedule obtained with 5 percent achieves the best performance for 12×12 tile matrix in actual execution. Figure 3.15 shows actual execution trace, where we provide partition information based on this schedule. In this trace, we can see that communication is almost overlapped with computation, except at the end which is intrinsic to the Cholesky kernel, and all resources are utilized properly.

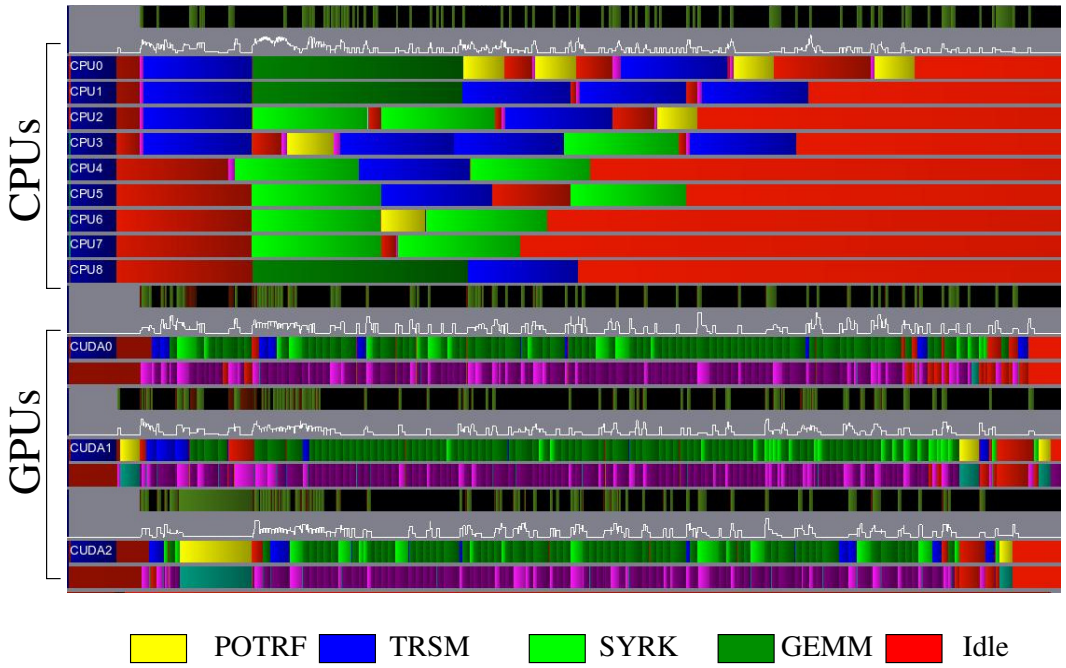


Figure 3.14: Actual execution trace based on HP+PCEPT schedule information - 12 × 12 tile matrix.

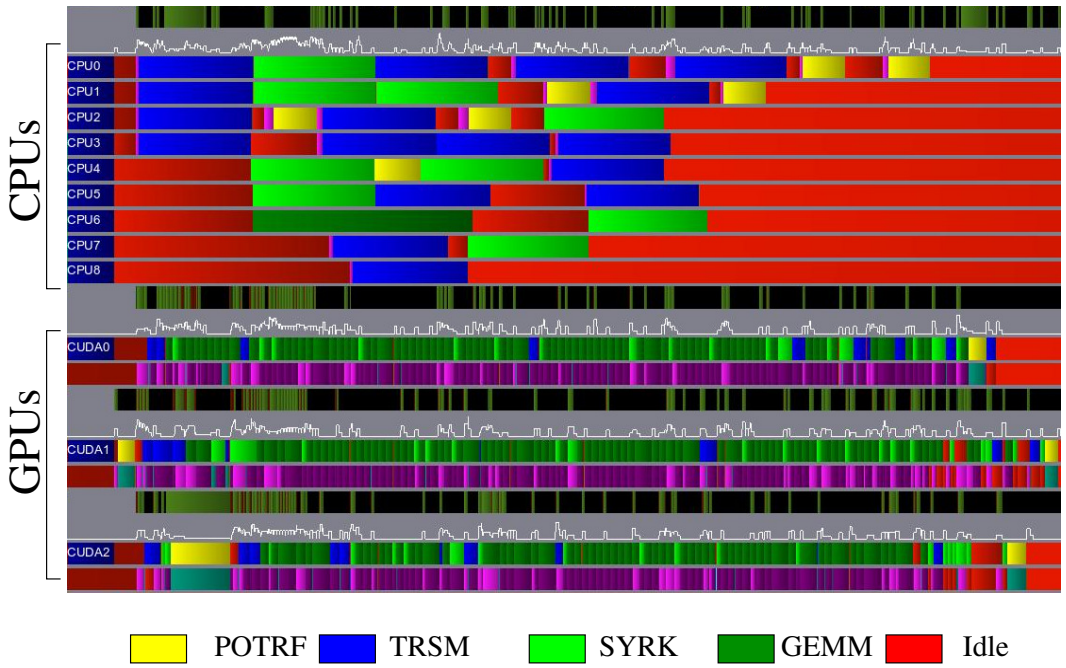


Figure 3.15: Actual execution trace based on modified HP+PCEPT schedule information - 12 × 12 tile matrix. Execution timings of all tasks on CPU are increased by 5 percent to calculate modified HP+PCEPT schedule.

3.10 Conclusion and Perspectives

This chapter aims at providing a fair comparison between static and dynamic scheduling strategies on heterogeneous platforms consisting of CPU and GPU nodes. Runtime dynamic schedulers make their decisions based on the state of machine, on the set of available tasks and possibly on task priorities computed online. The success of these dynamic strategies are motivated by expected weaknesses and limitations of static schedulers. First, it is well known that scheduling problems are hard (NP-Complete) and even hard to approximate with unrelated resources (what is the case in CPU-GPU platforms). Second, it has been observed that execution times of kernels in nodes where many resources (cache, memory, buses) are shared suffer high variance and it is generally assumed that the difficulty to predict execution times makes static schedulers useless. An original contribution of this chapter is to prove that this last assertion is in general not true and that static schedules (for Cholesky factorization) are in fact robust to variations in execution times. On the other hand, the consequence of the greedy nature of basic dynamic strategies is that they make a poor use of "slow" resources like CPUs. Since the overall processing power of CPUs is in general small, this does not hurt too much the GFlop/s performance of kernels. Nevertheless, we have proved that combining dynamic strategies with simulation in order to build less myopic algorithms can significantly improve their performance.

We have also considered a family of dynamic schedulers (HeteroPrio) that performs poorly on general graphs but greatly benefits from basic qualitative information about the task graph on platforms composed of CPUs and GPUs. We plan to generalize this class of schedulers for platforms consisting of several types of resources. We are also interested to study the theoretical performance of this class of schedulers.

In the longer term, this chapter opens many interesting perspectives. In particular, it advocates the design of efficient static schedules on heterogeneous unrelated machines and also advocates the introduction of as much static knowledge about the application as possible into dynamic schedulers in order to achieve good performance.

This page is intentionally left blank.

Chapter 4

Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources

In previous chapters, we proposed and analyzed a set of strategies by adding more static (resp. dynamic) features into dynamic (resp. static) strategies on platforms consisting of GPUs and CPUs. We exhibited that static-based strategies with dynamic corrections are very efficient, even in a context where performance estimations are not very good. We proposed different performance bounds of tasks graphs. We also proposed and evaluated a resource centric dynamic scheduling strategy, HeteroPrio, for task graphs on exactly two types of resources. However, this restriction can be limiting, for example on nodes with several types of accelerators, but not only this. Indeed, an interesting approach to increase resource usage is to group several CPU cores together, which allows to use intra-task parallelism. In this chapter, we propose a generalization of HeteroPrio to the case with several classes of heterogeneous workers. We provide extensive evaluation of this algorithm with Cholesky factorization, both through simulation and actual execution, compared with HEFT-based scheduling strategy, the state of the art dynamic scheduling strategy for heterogeneous systems. Experimental evaluation shows that our approach is efficient even for highly heterogeneous configurations and significantly outperforms HEFT-based strategy.

4.1 Introduction

Most runtime systems rely on a greedy HEFT-based scheduling strategy, where typically the highest priority ready task is allocated to the resource that is expected to complete it first, based on the estimation of the transfer time of input data and on the estimation of the execution time on the different resources [22, 104]. However, in presence of strongly heterogeneous resources,

it has been observed in Chapter 3 that such a scheduling policy tends to make poor use of slow resources. Indeed, when the acceleration ratio on the accelerators is high, all ready tasks tend to be allocated to accelerators, even though they are far from the critical path and could have been processed on a slow resource without hurting the overall execution time. Another dynamic strategy named HeteroPrio has been proposed [16] to cope with this problem, that relies on the affinities between tasks and resources. When properly tuned, this strategy has been proven to be more efficient than greedy strategies in presence of GPUs and CPUs for Cholesky factorization, in Chapter 3. Its main drawback is that it is limited to two types of resources, *i.e.* one sort of CPU and one sort of GPU. The main goal of this chapter is to extend such affinity based scheduling algorithms to any number of heterogeneous resources.

Such a generalization is obviously desirable to be able to handle platforms with more than one type of accelerator (with both GPU and Xeon Phi, for example). But it can also be very useful for more regular platforms, for the following reason. Many applications are parallelized using a uniform granularity: homogeneous block or tile decomposition where the choice of the tile size is a crucial parameter for performance. Indeed, a small granularity leads to poor performance on the GPU side, whereas large tiles decrease the parallelism available in the task graph, and dramatically increase the cost of bad load balancing decisions. Thus, the solution adopted by dense linear algebra libraries [17, 37, 84] is to compute a unique common size that represents the best trade-off. A more recent proposition is to relax this constraint either by splitting at runtime coarse grain tasks [102] or to aggregate CPU cores to process larger tasks [50]. Both approaches are equivalent to cluster CPUs together so as to build more powerful resources and to use parallel kernels on such CPU groups. This helps the scheduling algorithm since the composite platform is less heterogeneous: for instance, in the context of Cholesky factorization, the maximal heterogeneity ratio between a GPU and a large CPU group is less than 4 (and some kernels even execute faster on clustered CPU groups, see Table 4.1), what makes greedy scheduling algorithms more efficient, as advocated in [50]. For now, the clustering is determined statically for the duration of the whole execution, and there is a trade-off between lowering the critical path using groups of cores and increasing parallel efficiency by using many individual cores. In any case, the resulting platform appears to the runtime system as containing many different types of resources: individual CPUs, CPU groups of different sizes, and (possibly heterogeneous) GPUs.

In this chapter, we propose extensions of the affinity based scheduler that are suited to more than one type of resource and we demonstrate their efficiency on platforms consisting simultaneously of accelerators, several types of CPU groups and individual CPUs. However, the question of how to optimally build the groups, given the kernel, the size of the problem and the performance of individual resources, is out of the scope of this thesis. More specifically, this

chapter is organized as follows. Additional context and Related Works are presented in Section 4.2, the presentation of HeteroPrio and its adaptation to more than two types of resources is presented in Section 4.3. At last, the comparison between affinity based schedulers and HEFT based scheduler on both Cholesky and QR factorizations is presented in Section 4.4, before concluding remarks in Section 4.5.

4.2 Background and Related Work

When considering a task based application running on a heterogeneous system, a major challenge is related to the affinity between tasks and resources. This issue is particularly critical when designing dynamic schedulers for such systems. To illustrate this claim, we consider two dense factorization applications, namely Cholesky and QR. We report in Table 4.1a (resp. 4.1b) the performance for the different types of tasks composing the Cholesky (resp. QR) factorization. We can see that the GPU device is more suited for certain types of kernels (*e.g.* DGEMM, DTSMQR, etc.) than others. We can see also that for these kernels, the acceleration factors are large, what makes the platform strongly heterogeneous from the point of view of the scheduling algorithm. However, as mentioned in the Section 4.1, it is possible to reduce the heterogeneity of the platform by assigning a single task to a group of resources; this was introduced in [102] and [50]. We can observe in Table 4.1 that some kernels are very scalable (*e.g.* DGEMM, DSYRK, DTSMQR, etc.), some others have moderate scalability (*e.g.* DPOTRF, DORMQR) and finally some kernels exhibit poor scalability (*e.g.* DGEQRT and DTSQRT). We can also notice that when relying on medium to large CPU groups, the heterogeneity of the platform is strongly reduced: some kernels are even faster on the CPU group than on a GPU. Finally, since the scalability of the kernels is sublinear, it is better to rely on small groups of CPUs when the number of ready tasks is large enough. On the other hand, when the parallelism arising from the DAG is small, one may want to rely on large CPU groups. From the scheduling point of view, an adaptation of the HEFT algorithm to tackle the problem of dynamically scheduling parallel tasks was presented in [50].

On a more theoretical side, the work presented in this chapter is related to the theory of parallel tasks scheduling [58], in which each task can be assigned to a group of processors. There has been no study of parallel tasks for heterogeneous platforms, except very recently for independent tasks [30]; furthermore we are interested here in the case where the partition of processors into groups does not change during the execution of the application.

	DPOTRF	DTRSM	DSYRK	DGEMM
1 core (Gflop/s)	27.78	34.42	31.52	36.46
GPU / 1 core	1.72	8.72	26.96	28.80
10 cores / 1 core	5.55	6.75	6.90	7.77
5 cores / 1 core	4.20	4.50	4.66	4.49
2 cores / 1 core	1.88	1.95	1.93	1.94

(a) Cholesky factorization.

	DGEQRT	DORMQR	DTSQRT	DTSMQR
1 core (Gflop/s)	22.08	33.78	17.63	32.93
GPU / 1 core	1.91	15.90	1.87	14.64
10 cores / 1 core	1.65	4.10	0.73	6.94
5 cores / 1 core	1.67	3.30	1.25	4.05
2 cores / 1 core	1.33	1.77	1.16	1.91

(b) QR factorization (IB=128).

Table 4.1: Acceleration factors of Cholesky and QR factorization kernels normalized to the performance of one core with a tile of size 960.

4.3 Affinity Based Scheduling

As mentioned in the Section 4.1, a dynamic scheduling strategy named HeteroPrio, based on the affinities between tasks and resources, has been proposed in [16] for a set of independent tasks and extended for general task graphs in Chapter 3 in the case of GPUs and CPUs. In this section, after a brief presentation of the underlying principle of HeteroPrio, we propose a generalization to platforms with more than two types of resources.

4.3.1 Affinity Based Scheduling for Two Classes of Resources

We present the main ideas of HeteroPrio, and we refer the interested reader to Chapter 3 for a complete description of the algorithm. HeteroPrio relies on the acceleration ratios of tasks on GPUs to establish an affinity between the resources and the different types of tasks. In order to make the most out of the heterogeneous resources, GPUs should preferably execute tasks with higher acceleration factors, and CPUs should execute tasks with lower acceleration factors. To this end, HeteroPrio creates several queues, one for each type of tasks, which are ordered by acceleration factor and contain the list of ready tasks. When a CPU (resp. a GPU) becomes idle, it receives a task from the non empty queue with the lowest (resp. highest) acceleration factor. This algorithm was improved in several ways in Chapter 3. First, in order to avoid delaying tasks on (or close to) the critical path of the task graph, it is important to ensure that more critical tasks are executed on the GPUs. This is done

by sorting each ready queue by priority, computed as the distance to the exit node of the graph. GPUs are given the highest priority task from their queue, and CPUs are given the lowest priority task to ensure that urgent tasks are not delayed. This trade-off between affinity and priority is strengthened by another improvement: GPU queues with similar acceleration factors are merged, so that the algorithm focuses more on high priority tasks. As an example, let us consider the case of Cholesky factorization, with the task performance described in Table 4.1. In that case, HeteroPrio creates three queues for the GPUs, the first one regrouping DSYRK and DGEMM ready tasks, the second one containing DTRSM ready tasks, and the last one containing DPOTRF ready tasks. For the CPUs, HeteroPrio creates 4 ready queues containing ready DPOTRF, DTRSM, DSYRK and DGEMM tasks respectively, in that order.

Finally, a spoliation mechanism was added: whenever a GPU is idle while a DSYRK or DGEMM task is being executed on a CPU (for which it is badly suited), then the GPU restarts the execution of this task if it allows to finish that task earlier. In practice, stopping the execution of the kernels might be technically difficult, especially to enforce data coherency. However, the same behavior can be obtained by speculatively simulating the behavior of the algorithm before deciding to execute a task on a CPU, and if the task needs to be spoliated later, HeteroPrio decides to delay the execution of this task until a GPU becomes available. Alternatively, it is also possible to pre-compute (using simulation) a complete schedule with spoliation and to apply it on the real platform afterward.

4.3.2 Generalization to more than Two Classes of Resources

Generalizing Acceleration Factor Adapting this algorithm to the case of more than two types of resources is not straightforward, in particular because the central notion of acceleration factor does not make sense anymore in that case. It is thus necessary to identify a new way of deciding which tasks should be favored for execution on each of the given resources. In this section, we present two possible ways of computing scores which generalize the acceleration factor, and thus provide two different ways for the resources to favor different task types. The main principle of HeteroPrio remains unchanged, though: whenever a resource is free, it picks a ready task among the task types with the highest score.

The first scoring system is called `Area` because it relies on a generalization of the so-called *area bound* in the homogeneous case. The idea is to compute the allocation of tasks that minimizes the overall execution time when ignoring dependencies and assuming that all processors work without idle time. This allocation can be obtained by solving a small scale linear program (described in Chapter 2), and it provides a generic way of detecting which tasks are more

suited to which resources. In the `Area` system, the score of task type t for resource r is simply the proportion of tasks t that resource r would perform in this idealized setting. In the case of two resources, the optimal proportions are assigned following the ordering by acceleration factors. Hence this scoring system generalizes the behavior of the original HeteroPrio.

The second scoring system is called the *Heterogeneity Index* (`Het.Index`), and is computed in the following way. Let us denote by T the set of task types, by R the set of resources, and by $E(t, r)$ the execution time of task t on resource r. Let us consider for every task type t the maximum execution time $E_{\max} = \max_{i \in R} E(t, i)$ and the minimum execution time $E_{\min} = \min_{i \in R} E(t, i)$. We define $\texttt{Het.Index}(t, r) = \frac{E_{\max} \times E_{\min}}{E(t,r)^2} = \frac{E_{\max}}{E(t,r)} \times \frac{E_{\min}}{E(t,r)}$, and we use $\texttt{Het.Index}(t, r)$ as a score to decide which task type to favor for resource r. The idea behind this definition is that the first term $\frac{E_{\max}}{E(t,r)}$ represents how "good" this resource is compared to the worst possible one, and the second term represents how "bad" it is compared to the best one. This score is also a generalization of the acceleration factor: with GPUs and CPUs only, the heterogeneity index of GPUs is equal to the acceleration factor, and for CPUs, it is equal to the inverse of the acceleration factor.

Other considerations As mentioned above, it is important to take task priorities into account, by making sure that "fast" resources are given high priority tasks. Characterizing "fast" resources is straightforward in the case with only two resources, because GPUs are always faster than a single core. To generalize this to the multi-resource case, we propose the following approach.

For each resource r, we compute the geometric mean μ_r of the execution timings of all tasks on that resource ($\mu_r = \left(\prod_{t \in T} E(t, r)\right)^{\frac{1}{|T|}}$). This geometric mean measures the overall aggregated speed of resource r. We then compute the average (arithmetic mean) of these μ_r, and we classify a resource as "fast" if its value μ_r is below the average, and as a "slow" resource otherwise. "Fast" resources are given high priority tasks, and are allowed to perform spoliation on "slow" resources. Furthermore, as mentioned above, in HeteroPrio an emphasis is made on high priority tasks by merging queues with similar acceleration factors on GPUs. We generalize this on fast resources, by merging queues with similar scores. In practice, we have found that the best trade-off value for this parameter is to merge queues when the difference in score is below 25 %.

4.3.3 An Example with Both Scoring Systems

To understand the working principle of both scoring systems (`Area` and `Het.Index`), and to exhibit their difference, let us consider multiple instances of two types of tasks (T1 and T2) on three types of resources (R1, R2 and R3). Table 4.2 shows execution timings of both types of tasks on all resources. It also shows

	T1	T2
R1	100	200
R2	120	60
R3	200	75

(a) Execution Timings

	T1	T2
R1	60	0
R2	40	20
R3	0	80

(b) `Area` score

	T1	T2
R1	2.0	0.3
R2	1.4	3.3
R3	0.5	2.1

(c) `Het.Index` score

Table 4.2: Execution timings, Area and Het.Index scores on different resources for different types of tasks.

`Area` (Table 4.2b) and `Het.Index` (Table 4.2c) scores for both tasks on all resources.

On resource R1, for both scoring systems, the score of task T1 is higher than the score of task T2, therefore R1 will prefer tasks of type T1 in both scoring systems. Similarly, task T2 has higher score than T1 on resource R3, and therefore resource R3 will prefer tasks of type T2 in both scoring systems. On the other hand, in the `Area` scoring system, resource R2 will prefer task type T1 but `Het.Index` will pick in reverse order (prefer task type T2) due to higher `Het.Index` value for task type T2.

4.4 Experiments and Results

To evaluate the behavior of proposed scheduling heuristics, we present a set of experiments to assess the interest of our approach. First of all, we consider a platform composed of two Haswell Intel® Xeon® E5-2680 processors having 12 cores each and four Nvidia K40-M GPUs (this platform is different from what we considered in previous chapters). As mentioned in previous chapters, most runtime systems dedicate 1 CPU core to efficiently exploit each GPU. As a consequence, we can view our node as being composed of 20 CPU workers and 4 GPU workers. Throughout this chapter, all results are obtained with Intel icc and MKL version 2015.5.223 in addition with CUDA 7.0.28. We also ensure MKL_DYNAMIC flag is turned off to strictly control the number of used threads. Moreover, we consider a task-based implementation of two very common linear algebra operations (namely Cholesky and QR factorizations), which are decomposed in a number of basic kernels (see Figures 4.2a and 4.2b). These operations are implemented in the Chameleon [8] library running on top of the StarPU runtime system to assign tasks onto CPU cores or GPU devices. The experimental study is done in two steps: we first evaluate the different scheduling heuristics using simulation, and then we assess the performance of the best configurations in real-life executions. Note that we will consider both Cholesky and QR factorizations for the simulation case while we will only focus on the Cholesky kernel for the real-life case for the sake of simplicity.

4.4.1 Tuning of Tile Size Parameter

A crucial issue encountered when trying to exploit both CPUs and accelerators lies in the fact that these devices have very different characteristics and requirements. Compared to regular CPUs, a GPU for instance is composed of many lightweight cores and requires massive parallelism to hide memory latencies and thus to fully exploit its potential performance. As a result, GPUs typically exhibit better performance when executing kernels featuring numerous threads, which we call *coarse grain kernels* in the remainder of this chapter. On the other hand, regular CPU cores typically reach their peak performance with fine grain tasks working on a reduced memory footprint. To illustrate this claim, we provide in Figure 4.1 a performance profile of the matrix product kernel (DGEMM) on the two devices composing our experimental platform. We can observe that the sequential MKL implementation of the DGEMM kernel (for a regular CPU core) reaches its peak performance for matrix sizes greater than 200 while in the case of the cuBLAS kernel (for the GPU device), the GPU reaches its peak performance for sizes above 2000.

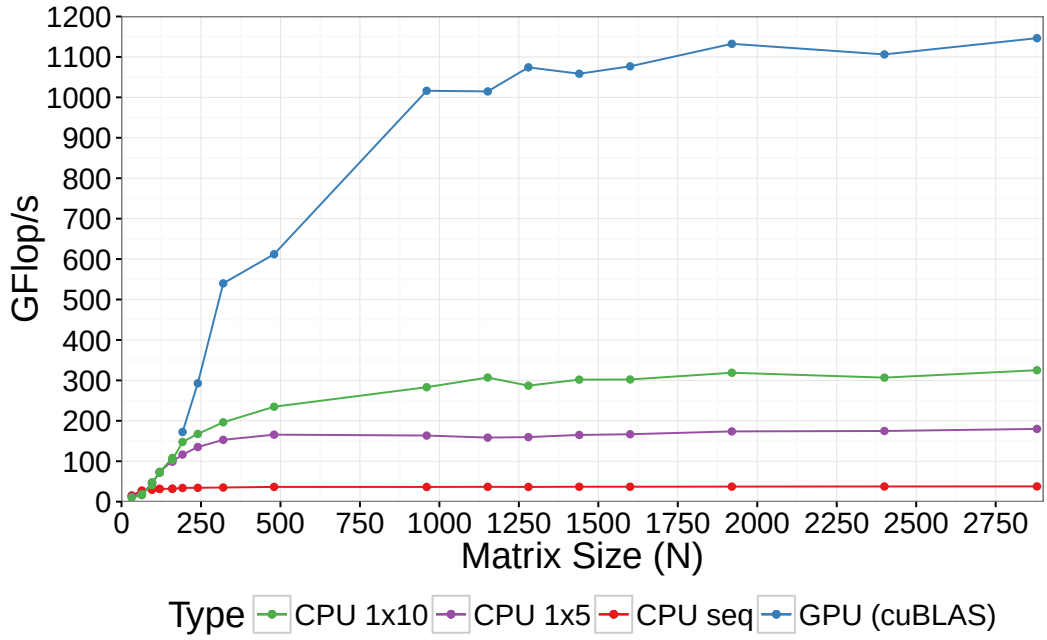


Figure 4.1: GEMM performance.

Unfortunately, runtime systems often consider accelerators as single devices, and treat individual cores equally. Because many applications are parallelized using homogeneous block or tile decomposition, runtime system schedulers have to cope with very different durations when executing tasks over single cores or over accelerators, resulting in situations where only a few tasks are assigned to CPUs because of bad scores computed by the performance

prediction-based heuristics. As a consequence, task-based applications running on such heterogeneous platforms typically adopt an intermediate granularity, chosen as a trade-off between coarse-grain and fine-grain tasks. A small granularity would indeed lead to poor performance on the GPU side, whereas large kernel sizes would dramatically increase the cost of wrong load-balancing decisions. This basic solution is used by state-of-the-art dense linear algebra libraries [17, 37, 85]. In the remainder of this chapter, we will use the same approach and consider a tile size of 960 that represents a good compromise in our context. Throughout this chapter, all matrix sizes are thus expressed in terms of number of tiles per row (or column).

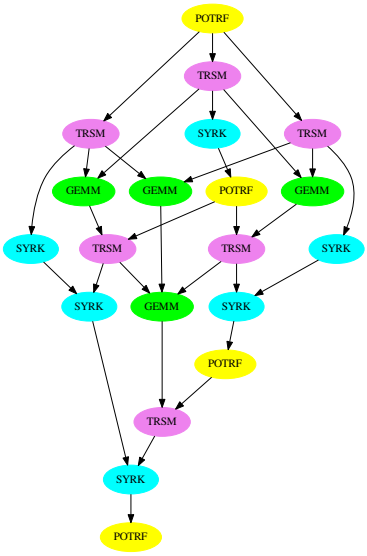
4.4.2 Experimental Framework

In this section, we present an analysis of the greedy HEFT-based strategy and of the two proposed variants of HeteroPrio (namely `Area` and `Het.Index`). Let us first describe the analysis and experimental methodology.

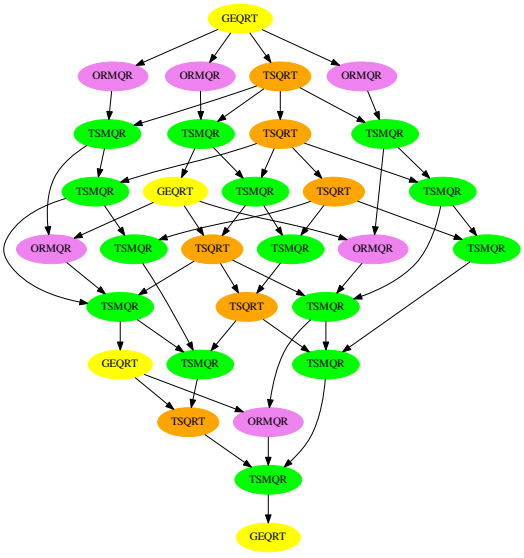
We rely on an adaptation of CHAMELEON which is able to process parallel tasks. This implementation does not change algorithms and subsequent DAGs. When a parallel kernel needs to be called (relying on Intel PARALLEL MKL), we invoke a specific prologue function to ensure that it will use the right set of resources [50]. Thanks to the hwloc framework [40], we take into account the machine topology to cluster resources together so as to ensure a proximity between resources of the same group. We measured the execution time of each of the underlying kernels on the GPUs as well as on various number of CPUs (part of these measurements are depicted in Table 4.1).

We use these timings to perform simulations of the behavior of each considered scheduling algorithm on each task graph. To simplify the simulations, we assume that it is possible to overlap communications with computations, and we thus neglect communication costs. In order to explore a wide range of cases, we analyze all possible ways to group the 20 CPU cores in clusters of size at most 10 (on our platform, it is not efficient to use groups larger than 10 cores due to NUMA effects). This yields to 530 different configurations, and for each configuration, we compare the performance of each considered scheduling algorithm. The HEFT algorithm is implemented as described in Section 4.1, combining a prioritization of tasks by their distance to the exit node with a greedy strategy which allocates tasks so as to finish them as early as possible. For each configuration, we also compute an upper bound on the achievable performance (the Iterative Bound of Chapter 3.4), which is obtained by solving a preemptive relaxation of the problem, expressed as a (rational) linear program. This upper bound is stronger than the commonly used GEMM peak bound, and provides a good hint on how well the task graph is suited to each particular platform.

In addition, we also compare the Cholesky factorization performance in



(a) Cholesky Factorization DAG



(b) QR Factorization DAG

Figure 4.2: Application task graphs for 4 ×4 tile matrix.

actual executions. For the HEFT based algorithm, we use the implementation available in StarPU which is based on the minimum completion time heuristic to schedule tasks on computational unit – thus a representative of state of the art HEFT heuristic. For HeteroPrio, in order to ease the implementation of the spoliation feature, we compute an offline HeteroPrio schedule in simulation

mode and run this schedule with StarPU runtime system in real execution, with dynamic adaptations discussed in Section 4.4.4.

4.4.3 Simulation Results & Analysis

The obtained results are shown in Figure 4.3 for Cholesky factorization, and in Figure 4.4 for QR factorization. Each column represents a given scheduling algorithm, and each row corresponds to a matrix size, expressed as the number N of tiles of size 960 in each row or column. Each dot corresponds to one given configuration, with the y axis showing the obtained performance, expressed in GFlop/s. The x axis represents the number of clusters in each configuration: this goes from 2 for the configuration with 2 groups of 10 cores, to 20 for the configuration with 20 single-core clusters. Configurations with small number of clusters thus have larger clusters, and correspond to the instances where the heterogeneity of the whole platform is lower (since CPU clusters achieve performance close to the one of a GPU). On the other hand, configurations with a larger number of clusters are more heterogeneous. For each scheduling algorithm, a horizontal line shows the performance with individual CPUs and GPUs (without CPUs clustering, *i.e.* 20 clusters) and acts as a reference line for performance comparison.

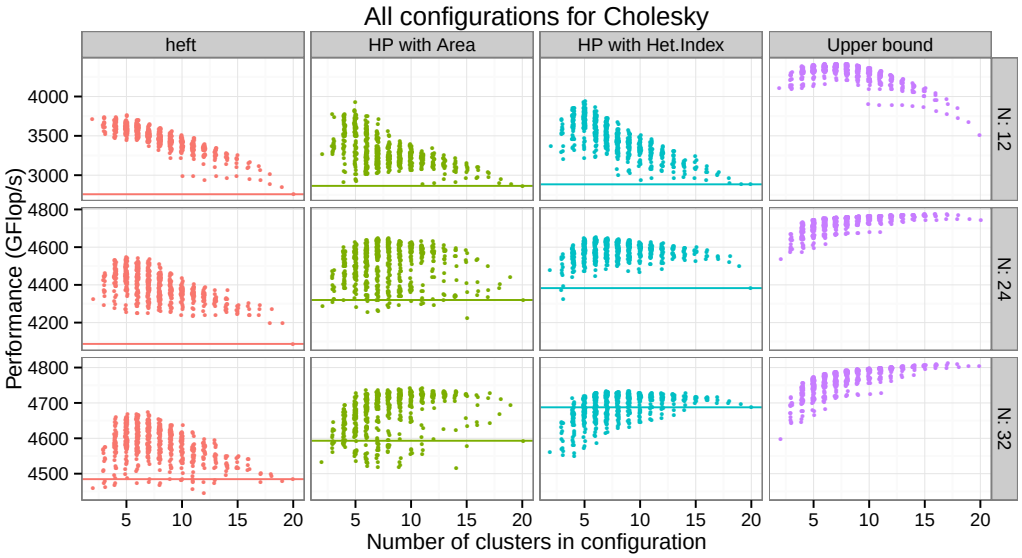


Figure 4.3: Performance results for all configurations for Cholesky factorization.

For Cholesky factorization (Figure 4.3), we can make the following observations. HeteroPrio variants performance is better than heft performance for

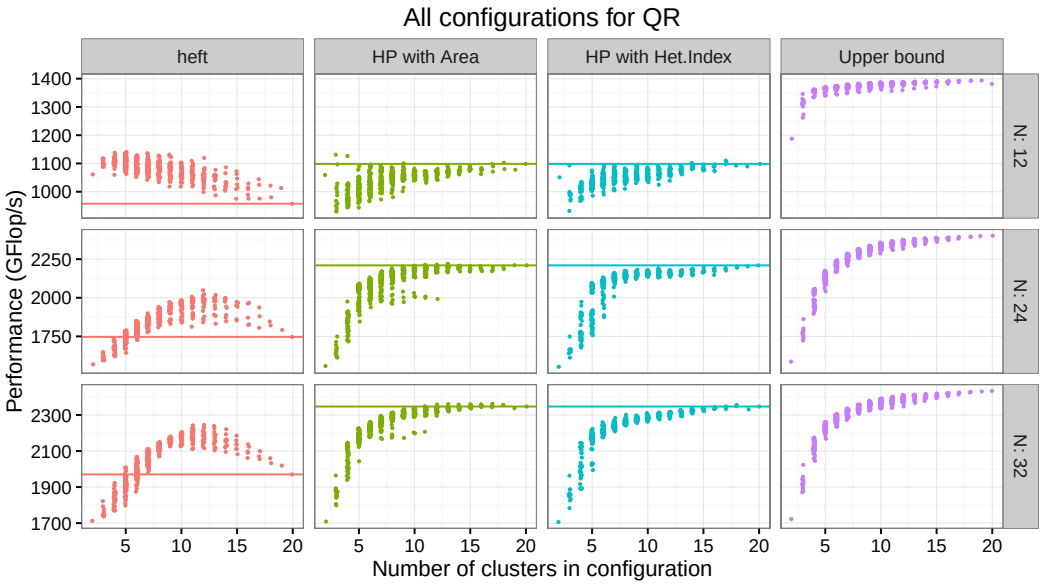


Figure 4.4: Performance results for all configurations for QR factorization.

all considered matrix sizes as expected. The heft algorithm requires a relatively small number of clusters to achieve good performance, showing that this algorithm does not cope well with strong platform heterogeneity, even for large matrix sizes. On the other hand, the HeteroPrio variants are able to make good use of heterogeneous configurations, except for very small matrix sizes. In the case of very small matrix sizes, it is however worth noting that the upper bound on performance does also drop for a large number of clusters, which hints that this performance drop is intrinsic to the task graph: the performance in that case is limited by the critical path of the graph, and clustering CPUs is necessary to obtain good performance. In general however, the upper bound is not enough to predict which configuration will provide the best performance for the algorithms. We can also notice that the performance of the `Het.Index` is more stable than `Area` for all matrix sizes, and there is a large number of configurations for which `Area` achieves significantly lower performance than `Het.Index`; however, their best-case performance is comparable. We can explain this better performance for the `Het.Index` variant by the following reason. The `Area` score is based on a global view of the task graph without dependencies and provides an overall repartition of the tasks. This repartition would be perfect if all tasks were independent, but the ideal repartition actually changes over time as dependencies unfold. Additionally, for each resource, the optimal repartition often involves ties between types of tasks. For example, two types of tasks which are not well suited to a resource would be assigned a score 0 since 0% of these types of tasks should be executed

on this resource. The scheduler thus treats both types of tasks indifferently, whereas one may be much more inefficient than the other, and this results in slightly lower performance than `Het.Index` variant in some cases. It can also be observed that clustering CPU cores is not always beneficial and some configurations achieve lower performance than the reference performance, which indicates that performance is dependent on critical tasks as well as on task efficiency.

Similar observations can be made for QR factorization (Figure 4.4). A notable difference is the behavior of all scheduling algorithms (even the upper bound) when the number of groups is too low, where the performance drops strongly. This is due to the fact that the basic kernels used in QR factorization cannot be parallelized as efficiently as those used in Cholesky. Obtaining good performance in heft thus requires precise tuning on the group size to obtain configurations which have both low heterogeneity and small enough clusters. On the other hand, the good behavior of both variants of HeteroPrio with a large number of clusters enables to achieve good performance even in this case.

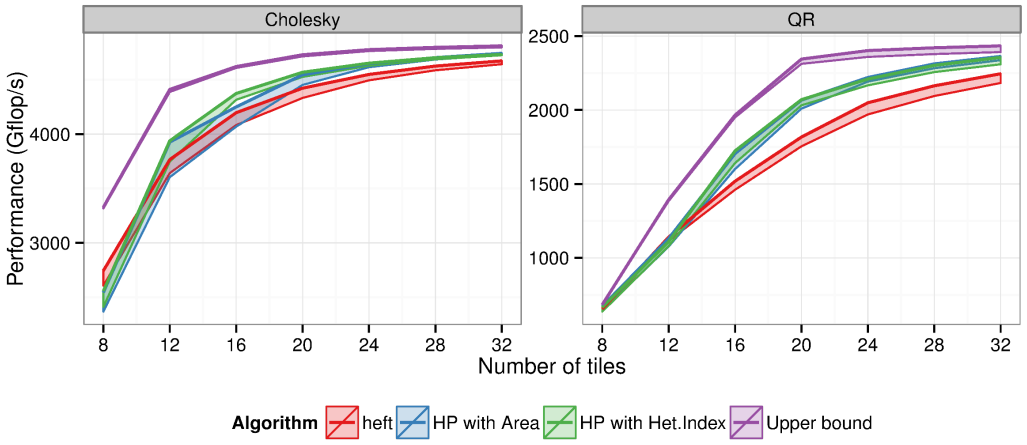


Figure 4.5: Performance of the 10% best configurations for both kernels.

In Figure 4.5, we present another view of the same results: this graph has been obtained by identifying the 10% best configurations for each matrix size and each algorithm. The graph shows the performance obtained on these configurations with a ribbon for each algorithm, where the highest point represents the best configuration, and the lowest point represents the worst among the 10% best configurations. This graph is meant to highlight the performance that can be obtained by each algorithm if the configuration can be adapted to the algorithm. This shows clearly that, for Cholesky, the gap between heft and HeteroPrio is wider for medium-size matrices, whereas for QR, the gap

is still present even for larger matrices. The `Area` and `Het.Index` variants have very similar best-case performance, except for Cholesky factorization of medium-size matrices, where `Het.Index` achieves best performance. Finally, the results obtained by HeteroPrio are reasonably close to the upper bound, in all considered cases.

In summary, HeteroPrio variants significantly outperform standard HEFT in all cases, and `Het.Index` variant is preferable due to a better overall stability. These results also highlight the benefits of CPU clustering: except for QR factorization of very large matrices, where the kernels have lower scalability and the best performance is always achieved with configurations that contain large number of CPU clusters.

4.4.4 Analysis of Actual Execution Traces

We now present results obtained in actual execution with the StarPU runtime, for the Cholesky factorization. As candidates for actual executions, we consider the configurations for which HeteroPrio achieves the best performance in simulation. For a given configuration, we build clusters based on locality information, thanks to hwloc [40], making sure that clusters do not cross the NUMA boundaries of the physical machine. From the HeteroPrio schedule, we obtain an allocation of tasks on resources, and an ordering of tasks for each resource, that we use for actual execution. Figure 4.6 shows the actual execution trace for a 24×24 matrix with the configuration and schedule for which HeteroPrio achieves the best performance in simulation. It exhibits a lot of small idle times on GPUs due to significant non-overlapping of data transfers with computations.

Since the communication costs are neglected in simulation, we propose the following two features to dynamically adapt the resulting static schedule to a different environment. First, whenever a CPU cluster lacks work (because no task assigned to it is ready yet), it can steal a task from another CPU cluster, preferably of similar size. Second, all tasks allocated to GPU are considered in a merged queue, from which tasks are assigned, in order, to the GPU which can finish it first. This allows to mitigate the number of data transfer operations among GPUs.

Figure 4.7 shows the actual execution trace for a 24 × 24 matrix with both above features implemented. It shows that most of the GEMM tasks are running on GPUs (last 4 resources in the trace) and communication is almost overlapped with computation for these tasks. However, before the tasks that run on CPU clusters (especially POTRF and TRSM tasks), a small idle time is introduced, due to data transfers, which cumulatively become significant and keep GPUs significantly idle in the end. To cope with this behavior, we propose to inflate the execution times considered in simulation for the CPUs, so as to take into account this communication overhead (similar to what we did in

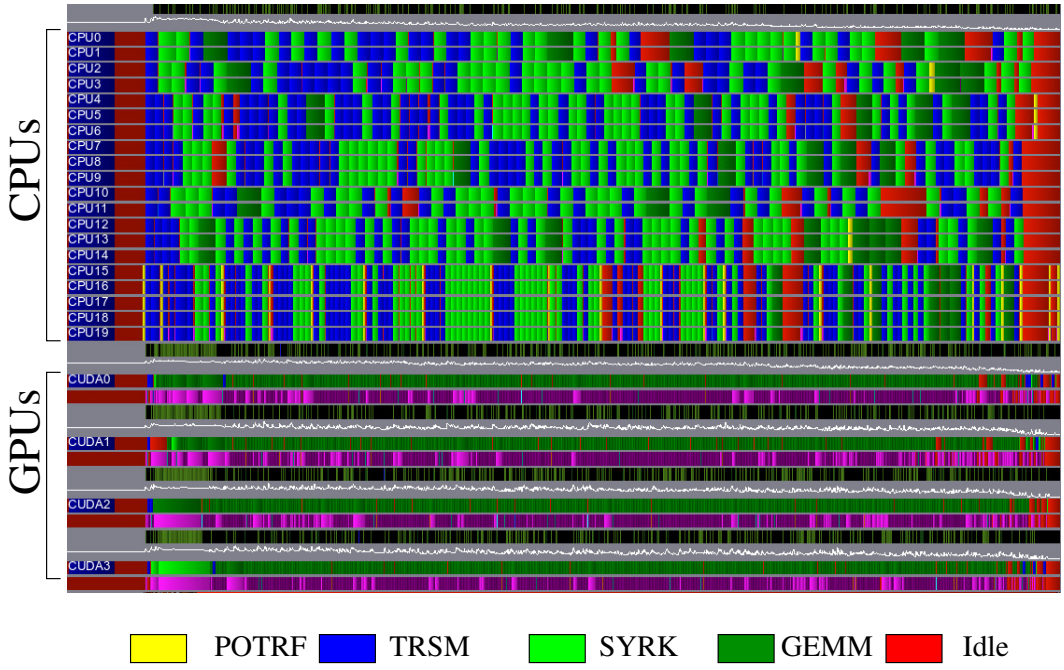


Figure 4.6: Execution trace for 24 × 24 with HeteroPrio schedule. Time is on the horizontal axis, resources are on the vertical axis, with GPUs at the bottom.

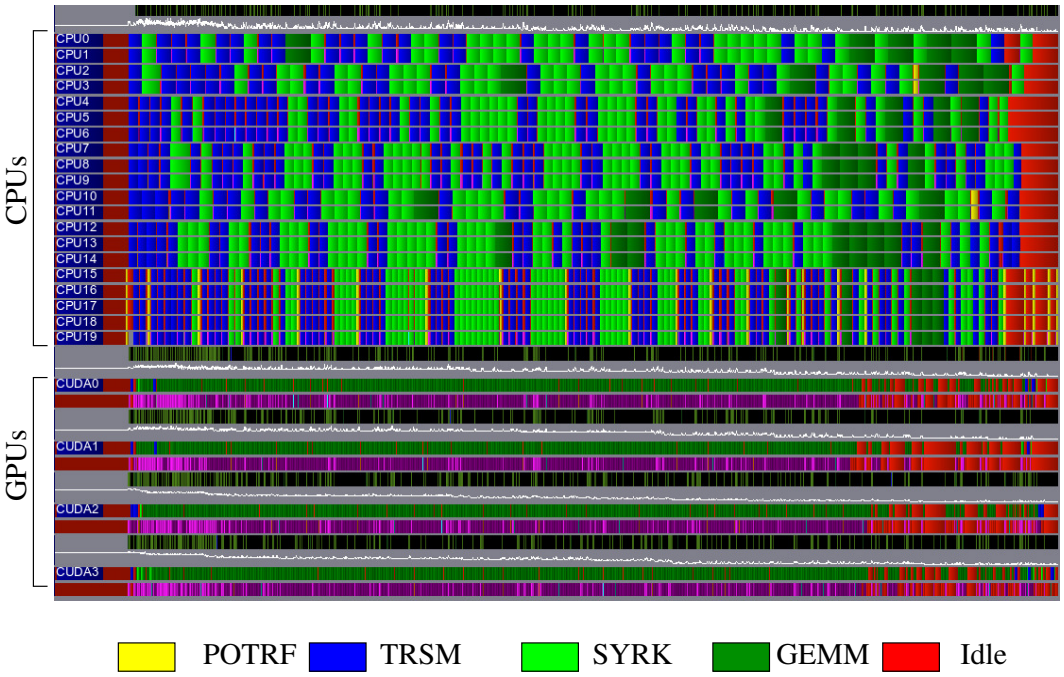


Figure 4.7: Execution trace with corrections for 24 × 24 with HeteroPrio schedule.

Chapter 3.9). We have tried different values, and observed that a 15 % increase in task execution times on CPU achieves the best load balancing among all workers in actual executions, for all matrix sizes. Figure 4.8 shows a real execution trace obtained with the HeteroPrio schedule with 15 % increment in CPU execution time of tasks. We can see that the load balancing is strongly improved: GPU devices and CPU cores are used until the very end of the execution. In the remainder of this section, we will use schedules obtained with 15 % increment in task execution times on CPUs.

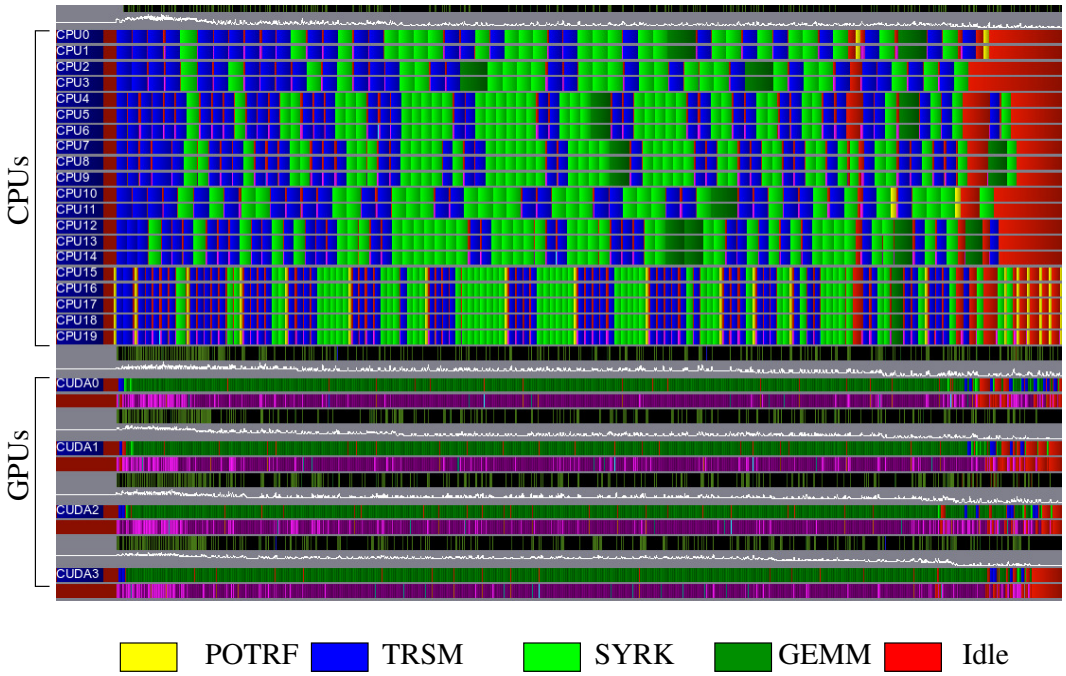


Figure 4.8: Load balanced execution trace for 24 × 24 with HeteroPrio schedule.

4.4.5 Actual Execution Performance Comparison

We compare the performance on Cholesky factorization, in actual execution and for different matrix sizes, of the different strategies considered in this chapter, together with MAGMA [17], a state of the art dense linear algebra library. We remind that HeteroPrio real execution (hp-best in Figure 4.9) comes from the execution of the best schedules obtained in simulation mode, with the two relaxations and the 15% correction as described in 4.4.4. For HEFT (heft-best in Figure 4.9), we use the cluster configuration that achieves the best performance in simulation with this strategy. We also run the HEFT scheduler with the configuration obtained for best HeteroPrio schedule, denoted with heft (hp-best config.) in Figure 4.9. In addition, we also provide the baseline

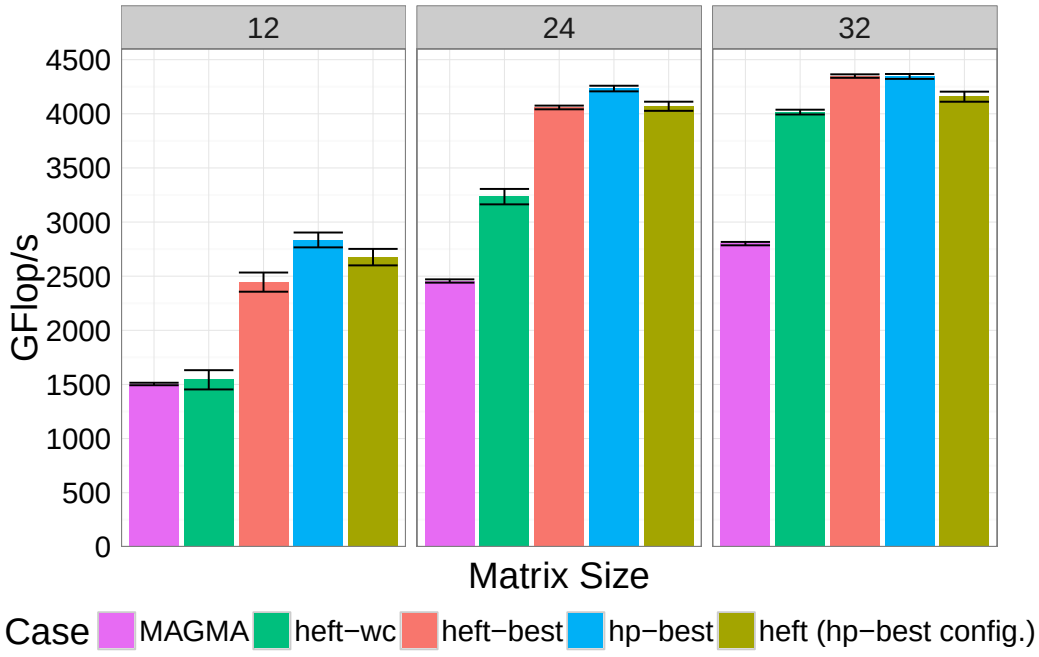


Figure 4.9: Performance results for the HEFT and HeteroPrio policies with selected configurations.

performance achieved by HEFT when not considering clusters of CPUs (hence each CPU core is used as a single worker), denoted as heft-wc, and the performance achieved by the MAGMA library. For each performance bar, we plot mean values and performance variation obtained from 10 runs.

For 12×12 matrix size, we observe that heft-wc and MAGMA achieve similar performance. When using clusters, with heft-best, the performance increases by 58% compared to heft-wc. This is expected since the amount of parallelism with such a matrix size is not enough to fill all 20 CPUs in heft-wc and results in bad performance. hp-best obtains 6% performance improvement over heft (hp-best config.), with the same cluster configuration. heft-best is showing slightly lower performance compared to heft (hp-best config.), which can be explained by the communications. For a low amount of tasks such as in this case, there are not enough tasks to fully overlap communications with computations and therefore the best cluster configuration identified through simulations may experience significant overhead due to non overlapped data transfers.

For large matrix sizes, we can observe that the gap between heft-wc and hp-best is reduced from 31% (for 24×24) to 8% (for 32×32). In addition, heft-best is more accurate and outperforms the other HEFT schedulers such as heft (hp-best config.). hp-best achieves a performance improvement of 4.5% over heft-best for 24×24. But hp-best does not achieve significant performance improvement over heft-best for 32×32, which is due to how the task allocation

evolves with increasing matrix size. Indeed, for larger matrices, the execution is mainly dominated by (almost independent) GEMM tasks, which makes the scheduling problem relatively easy and both hp-best and heft-best achieve almost the same performance. These results are consistent with Figure 4.5, which shows that the difference between HeteroPrio and HEFT is much smaller for 32×32 compared to the lower matrix sizes (and both are actually very close to the upper bound).

4.5 Conclusion

In this chapter, we present several extensions of the HeteroPrio scheduling strategy to the case with more than two types of resources. Besides the obvious case of platforms with different accelerator types, this capability is also crucial when CPU cores are clustered together to make use of intra-task parallelism, as it has been recently advocated in order to make a better use of all available resources and to build a more homogeneous platform. In order to assess the efficiency of our approach, we concentrate on Cholesky and QR factorizations although proposed techniques can easily adapt to other kernels or applications, provided that they are expressed as DAGs.

We perform extensive simulations and actual experiments on a heterogeneous platform composed of two Haswell Intel® Xeon® E5-2680 processors having 12 cores each and four Nvidia K40-M GPUs, using `StarPU`, a modern task-based runtime system. We show that HeteroPrio variants are able to make a very efficient use of almost all possible configurations of heterogeneous platforms. Together with the capability of clustering CPU cores, the heuristics that we propose allow to significantly improve the performance of task based applications.

In future works, we are planning to provide a complete dynamic implementation of HeteroPrio, so that such good performance can be obtained without relying on static schedules. In the longer term, this work opens many interesting perspectives, in particular about how to select the optimal configuration of CPU clusters, when the platform is too large for exhaustive search. It would also be interesting to study whether the performance can be improved by changing the clustering of CPUs during the execution instead of using the same configuration from the beginning to the end.

Chapter 5

HeteroPrio Approximation Ratios on Two Types of Resources

In previous chapters, we proposed different performance bounds for task graphs. We proposed and analyzed a set of strategies by adding more static (resp. dynamic) features into dynamic (resp. static) strategies on platforms consisting of GPUs and CPUs. We also extended and evaluated a new class of scheduling algorithm, HeteroPrio, which is based on affinity between tasks and resources, on exactly two types of resources, for general task graphs with very interesting results. We generalized it later to the case with several classes of heterogeneous resources. In this chapter, we provide a theoretical insight on the performance of HeteroPrio on two types of unrelated resources, by proving approximation bounds compared to the optimal schedules in the case where all tasks are independent and for different platform sizes. Interestingly, this shows that spoliation allows to prove bounded approximation ratios for a list scheduling algorithm on two unrelated resources, which is impossible otherwise. We also establish that almost all our bounds are tight. Additionally, we provide an experimental evaluation of HeteroPrio on real task graphs from dense linear algebra computations, which highlights the reasons explaining its good practical performance.

5.1 Introduction

As mentioned in Chapter 1, Most runtime systems such as StarPU [22], StarSs [82], SuperMatrix [46], QUARK [104], XKaapi [67] or PaRSEC [36] model the application as a DAG, where nodes correspond to tasks and edges to dependencies between these tasks. At runtime, the scheduler knows (i) the state of the different resources (ii) the set of tasks that are currently processed by all non-idle resources (iii) the set of (independent) tasks whose all dependencies have been solved (iv) the location of all input data of all tasks (v) possibly an estimation of the duration of each task on each resource and of each communication

between each pair of resources and (vi) possibly priorities associated to tasks that have been computed offline. Therefore, the scheduling problem consists in deciding, for an independent set of tasks, given the characteristics of these tasks on the different resources, where to place and to execute them. This chapter is devoted to this specific problem.

On the theoretical side, several solutions have been proposed for this problem, including PTAS (see for instance [61] for a recent contribution with accelerators). Nevertheless, in the target application, dynamic schedulers must take their decisions at runtime and are themselves on the critical path of the application. This reduces the spectrum of possible algorithms to very fast ones, whose complexity to decide which task to execute next should be very small (constant, linear or linearithmic in the number of ready tasks). Otherwise, scheduling overhead would be too high, and that would inhibit the application progress and may result in very poor performance.

Several scheduling algorithms have been proposed in this context and can be classified in several classes. The first class of algorithms is based on (variants of) HEFT [99], where the priority of tasks is computed based on their expected distance to the last node, with several possible metrics to define the expected durations of tasks (given that tasks can be processed on heterogeneous resources) and data transfers (given that input data may be located on different resources). To the best of our knowledge there is not any approximation ratio for this class of algorithms on unrelated resources and Bleuse et al. [31] have exhibited an example on m CPUs and 1 GPU where HEFT algorithm achieves a makespan $\Omega(m)$ times worse the optimal. The second class of scheduling algorithms is based on more sophisticated ideas that aim at minimizing the makespan of the set of ready tasks (see for instance [31]). In this class of algorithms, the main difference lies in the compromise between the quality of the scheduling algorithm (expressed as its approximation ratio when scheduling independent tasks) and its cost (expressed as the complexity of the scheduling algorithm). At last, a third class of algorithms has recently been proposed (see for instance Chapter 3), in which scheduling decisions are based on the affinity between tasks and resources, *i.e.* try to process a task on the best suited resource for it.

In this chapter, we concentrate on HeteroPrio that belongs to the third class and that is briefly described in Section 5.2. More specifically, we prove that HeteroPrio combines the best of all worlds. Indeed, after discussing the related work in Section 5.3 and introducing notations and general results in Section 5.4, we first prove that contrarily to HEFT variants, HeteroPrio achieves a bounded approximation ratio in Section 5.5 and we provide a set of proved and tight approximation results, depending on the number of CPUs and GPUs in the node. At last, we provide in Section 5.6 a set of experimental results proving that, besides its very low complexity, HeteroPrio achieves a better performance than the other schedulers based either on HEFT or on a more sophisticated ap-

	DPOTRF	DTRSM	DSYRK	DGEMM
CPU time / GPU time	1.72	8.72	26.96	28.80

Table 5.1: Acceleration factors for Cholesky kernels (size 960)

proximation algorithm for independent tasks scheduling. Concluding remarks are given in Section 5.7.

5.2 HeteroPrio Principle

5.2.1 Affinity Based Scheduling

HeteroPrio has been proposed in the context of task-based runtime systems responsible for allocating tasks onto heterogeneous nodes typically consisting of a few CPUs and GPUs [16].

As mentioned in previous chapters, in most runtime systems, tasks are ordered by priorities (computed offline) and the highest priority ready task is allocated on the resource that is expected to complete it first, given the estimation of the transfer times of its input data and the expected processing time of this task on this resource. These systems have shown some limits in strongly heterogeneous and unrelated systems, what is typically the case of nodes consisting of both CPUs and GPUs (see for instance Chapter 3). Indeed, the relative efficiency of accelerators, that we call the affinity in what follows, strongly differs from one task to another. Let us for instance recall the case of Cholesky factorization, where 4 types of tasks (kernels DPOTRF, DTRSM, DSYRK and DGEMM) are involved. The acceleration factors of the different Cholesky tasks have been presented in Chapter 4 and also depicted here in Table 5.1. In all what follows, acceleration factor is always defined as the ratio between the processing time on a CPU and on a GPU, so that the acceleration factor may be smaller than 1. From this table, we can extract the main features that will influence our model. The acceleration factor strongly depends on the kernel. Table 5.1 exhibits that some kernels, like DSYRK and DGEMM are almost 30 times faster on GPUs, DPOTRF is only slightly accelerated. Based on this observation, a different class of runtime schedulers for task based systems has been developed, in which the affinity between tasks and resources plays the central role. HeteroPrio belongs to this class. In these systems, when a resource becomes idle, it selects among the ready tasks the one for which it has a maximal affinity. For instance, in the case of Cholesky factorization, among the ready tasks, CPUs will prefer DPOTRF to DTRSM to DSYRK to DGEMM and GPUs will prefer DGEMM to DSYRK to DTRSM to DPOTRF.

HeteroPrio allocation strategy has been studied in the context of StarPU for several linear algebra kernels and it has been proved experimentally that

it enables to achieve a better utilization of slow resources than other strategies based on the minimization of the completion time (see Chapters 3 and 4). Nevertheless, in order to be efficient, HeteroPrio must be associated to a spoliation mechanism. Indeed, in the above description, nothing prevents the slow resource to execute a task for which it can be arbitrarily badly suited, thus leading to arbitrarily bad results. Therefore, when a fast resource is idle and would be able to restart a task already started on a slow resource and to finish it earlier than on the slow resource, then the task is spoliated and restarted on the fast resource. Note that this mechanism does not correspond to preemption since all the progress made on the slow resource is lost. It is therefore less efficient than preemption but it can be implemented in practice (see Chapter 3.7.4), what is not the case of preemption on heterogeneous resources like CPUs and GPUs.

In what follows, since task based runtime systems see a set of independent tasks, we will concentrate on this problem and we will prove approximation ratios for HeteroPrio under several scenarios for the composition of the heterogeneous node (namely 1 GPU and 1 CPU, 1 GPU and several CPUs and several GPUs and several CPUs).

5.2.2 HeteroPrio Algorithm for a set of Independent Tasks

Algorithm 3: The HeteroPrio Algorithm for a set of independent tasks.

```
Sort Ready tasks in queue Q by non-increasing acceleration factors
while all tasks did not complete do
    if all workers are busy then
        continue
    end if
    Select an idle worker W
    if Q ≠ ∅ then
        Remove a task T from beginning of Q if W is a GPU worker
        otherwise from end of Q
        W starts processing T
    else
        Consider tasks running on the other type of resource in decreasing
        order of their expected completion time. If the expected completion
        time of T running on a worker W' can be improved on W, T is
        spoliated and W starts processing T.
    end if
end while
```

Algorithm 3 describes the precise version of HeteroPrio which we consider in this chapter. Note that in Chapter 3, we were considering tasks in decreasing

order of their priorities to perform spoliation. But here we consider tasks in decreasing order of their expected completion time (Line 11 of Algorithm 3), which allows us to prove bounded approximation ratios for HeteroPrio. Also in previous chapters, for HeteroPrio, we proposed to create one ready queue per type of task but the same behavior can be achieved by a single ready queue as shown in Line 1 of Algorithm 3.

When priorities are associated with tasks then Line 1 of Algorithm 3 takes them into account for breaking ties among tasks with the same acceleration factor and put highest (resp. lowest) priority task first in the scheduling queue for acceleration factor≥ 1 (resp. < 1). Approximation factors proved in this chapter do not depend on how ties are broken and thus not on task priorities. However, considering task priorities allows schedulers to achieve good performance for task graphs (see previous chapters).

Queue of ready tasks in Algorithm 3 can be implemented as a heap. Therefore, time complexity of Algorithm 3 on m CPUs and n GPUs would be $\mathcal{O}(Nlog(N))$, where N is the number of ready tasks.

5.3 Related Works

The problem considered in this chapter is a special case of the standard unrelated scheduling problem $R||C_{\max}$. Lenstra et al [73] proposed a PTAS for the general problem with a fixed number of machines, and a 2-approximation algorithm, based on the rounding of the optimal solution of the linear program which describes the preemptive version of the problem. This result has recently been improved [95] to a $2 - \frac{1}{m}$ approximation. However, the time complexity of these general algorithms is too high to allow using them in the context of runtime systems.

The more specialized case with a small number of types of resources has been studied in [34] and a PTAS has been proposed, which also contains a rounding phase whose complexity makes it impractical, even for 2 different types of resources. Greedy approximation algorithms for the online case have been proposed by Imreh on two different types of resources [70]. These algorithms have linear complexity, however most of their decisions are based on comparing task execution times on both types of resources and not on trying to balance the load. The result is that in the practical test cases of interest to us, almost all tasks are scheduled on the GPUs and the performance is significantly worse. Finally, Bleuse et al [31, 29] have proposed algorithms with varying approximation factors ($\frac{4}{3}$, $\frac{3}{2}$ and 2) based on dynamic programming and dual approximation techniques. These algorithms have better approximation ratios than the ones proved in this chapter. But their time complexity is high which restricts their implementations in most state-of-the-art runtime systems. Furthermore, as we show in Section 5.6, their actual performance is not

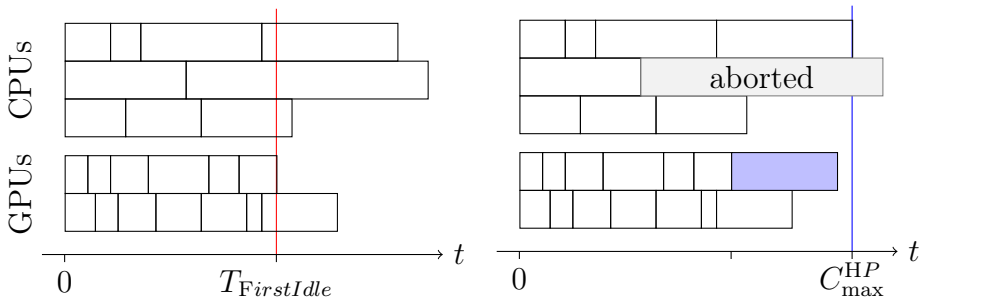
as good when used iteratively on the set of ready tasks in the context of task graph scheduling. We also show that HeteroPrio performs better on average than above mentioned algorithms, despite its higher worst case approximation ratio.

In homogeneous scheduling, list algorithms (*i.e.* algorithms that never leave a resource idle if there exists a ready task) are known to have good practical performance. In the context of heterogeneous scheduling, it is well known that list scheduling algorithms cannot achieve an approximation guarantee. Indeed, even with two resources and two tasks, if one resource is much slower than the other, it can be arbitrarily better to leave it idle and to execute both tasks on the fast resource. The HeteroPrio algorithm considered in this chapter is based on a list algorithm, but the use of spoliation (see Section 5.2.2) avoids this problem.

5.4 Notations and First Results

5.4.1 General Notations

In this chapter, we study the theoretical guarantees of HeteroPrio for a set of independent tasks. In the scheduling problem that we consider, the input is thus a platform of n GPUs and m CPUs and a set $\mathcal{I}$ of independent tasks, where task T_i has processing time p_i on CPU and q_i on GPU, and the goal is to schedule those tasks on the resources so as to minimize the makespan. We define the acceleration factor of task T_i as $\rho_i = \frac{p_i}{q_i}$. $C^{Opt}_{\max}(\mathcal{I})$ is used throughout this chapter to denote the optimal makespan of set $\mathcal{I}$.



(a) Schedule $\mathcal{S}^{NS}_{HP}$ (b) Schedule $\mathcal{S}_{HP}$, with one spoliated task

Figure 5.1: Example of a HeteroPrio schedule.

To analyze the behavior of HeteroPrio, it is useful to consider the list schedule obtained before any spoliation attempt. We will denote this schedule by $\mathcal{S}^{NS}_{HP}$, and the final HeteroPrio schedule is denoted by $\mathcal{S}_{HP}$. Figure 5.8 shows $\mathcal{S}^{NS}_{HP}$ and $\mathcal{S}_{HP}$ for a set of independent tasks $\mathcal{I}$. We define $T_{FirstIdle}$ as the first time any worker is idle in $\mathcal{S}^{NS}_{HP}$, this is also the first time any spoliation can

occur. Therefore after time $T_{FirstIdle}$, each worker executes at most one task in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. Finally, we define $C_{\max}^{HP}(\mathcal{I})$ as the makespan of $\mathcal{S}_{\text{HP}}$ on instance $\mathcal{I}$.

5.4.2 Area Bound

In this section, we recall *AreaBound*, a lower bound on the optimal makespan from Chapter 2, and characterize its different features. This lower bound is obtained by assuming that tasks are divisible, *i.e.* can be processed in parallel on any number of resources. More specifically, any fraction x_i of task T_i is allowed to be processed on CPUs, and this fraction overall consumes CPU resources for $x_i p_i$ time units. Then, the lower bound $AreaBound(\mathcal{I})$ for a set of tasks $\mathcal{I}$ on m CPUs and n GPUs is the solution (in rational numbers) of the following linear program.

Minimize $AreaBound(\mathcal{I})$ such that

$$\sum_{i \in I} x_i p_i \leq m \cdot AreaBound(\mathcal{I}) \tag{5.1}$$

$$\sum_{i \in I} (1 - x_i) q_i \leq n \cdot AreaBound(\mathcal{I}) \tag{5.2}$$

$$0 \leq x_i \leq 1$$

Since any valid solution to the scheduling problem can be converted into a solution of this linear program, it is clear that $AreaBound(\mathcal{I}) \leq C_{\max}^{Opt}(\mathcal{I})$. Another immediate bound on the optimal is $\forall T \in \mathcal{I}, \min(p_T, q_T) \leq C_{\max}^{Opt}(\mathcal{I})$. By contradiction and with simple exchange arguments, one can prove the following two lemmas.

Lemma 5.1. *In the area bound solution, the completion time on each class of resources is the same, i.e. constraints* (5.1) *and* (5.2) *are both equalities.*

Proof. Let us assume that one of the inequality constraints of area solution is not tight. Without loss of generality, let us assume that Constraint 5.1 is not tight. Then some load from the GPUs can be transferred to the CPUs which in turn decreases the value of $AreaBound(\mathcal{I})$. This achieves the proof of the Lemma 5.1. □

Lemma 5.2. *In $AreaBound(\mathcal{I})$, the assignment of tasks is based on the acceleration factor, i.e. $\exists k > 0$ such that $\forall i, x_i < 1 \Rightarrow \rho_i \geq k$ and $x_i > 0 \Rightarrow \rho_i \leq k$.*

Proof. Let us assume $\exists(T_1,T_2)$ such that (i) T_1 is partially processed on GPUs (*i.e.* , $x_1 < 1$), (ii) T_2 is partially processed on CPUs (*i.e.* , $x_2 > 0$) and (iii) $\rho_1 < \rho_2$.

Let WC and WG denote respectively the overall work on CPUs and GPUs in $AreaBound(\mathcal{I})$. If we transfer a fraction $0 < \epsilon_2 < min(x_2, \frac{(1-x_1)p_1}{p_2})$ of T_2

work from CPU to GPU and a fraction $\frac{\epsilon_2 q_2}{q_1} < \epsilon_1 < \frac{\epsilon_2 p_2}{p_1}$ of T_1 work from GPU to CPU, the overall loads WC' and WG' become the following.

$$WC' = WC + \epsilon_1 p_1 - \epsilon_2 p_2$$
$$WG' = WG - \epsilon_1 q_1 + \epsilon_2 q_2$$

Since $\frac{p_1}{p_2} < \frac{\epsilon_2}{\epsilon_1} < \frac{q_1}{q_2}$, then both $WC' < WC$ and $WG' < WG$ hold true, and hence the $AreaBound(\mathcal{I})$ is not optimal. Therefore, $\exists$ a positive constant k such that $\forall i$ on GPU, $\rho_i \geq k$ and $\forall i$ on CPU, $\rho_i \leq k$. □

5.4.3 Summary of Approximation Results

This chapter presents several approximation results depending on the number of CPUs and GPUs. Table 5.2 presents a quick overview of the main results proven in Section 5.5.

(#CPUs,# GPUs)	Approximation ratio	Worst case ex.
(1,1)	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$
(m,1)	$\frac{3+\sqrt{5}}{2}$	$\frac{3+\sqrt{5}}{2}$
(m,n)	$2+\sqrt{2} \approx 3.41$	$2+\frac{2}{\sqrt{3}} \approx 3.15$

Table 5.2: Approximation ratios and worst case examples.

5.5 Proof of HeteroPrio Approximation Results

5.5.1 General Lemmas

The following lemma gives a characterization of the work performed by HeteroPrio at the beginning of the execution, and shows that HeteroPrio performs as much work as possible when all resources are busy. At any instant t, let us define $\mathcal{I}'(t)$ as the sub-instance of $\mathcal{I}$ composed of the fractions of tasks that have not been entirely processed at time t by HeteroPrio. Then, a schedule beginning like HeteroPrio (until time t) and ending like $AreaBound(\mathcal{I}'(t))$ completes in $AreaBound(\mathcal{I})$.

Lemma 5.3. *At any time $t \leq T_{FirstIdle}$ in $\mathcal{S}^{NS}_{HP}$,*

$$t + AreaBound(\mathcal{I}'(t)) = AreaBound(\mathcal{I})$$

Proof. HeteroPrio assigns tasks based on their acceleration factors. Therefore, at instant t, $\exists k_1 \leq k_2$ such that (i) all tasks (at least partially) processed on GPUs have an acceleration factor larger than k_2, (ii) all tasks (at least

partially) allocated on CPUs have an acceleration factor smaller than k_1 and (iii) all tasks not assigned yet have an acceleration factor between k_1 and k_2.

After t, $AreaBound(\mathcal{I}')$ satisfies Lemma 5.2, and thus $\exists k$ with $k_1 \leq k \leq k_2$ such that all tasks of $\mathcal{I}'$ with acceleration factor larger than k are allocated on GPUs and all tasks of $\mathcal{I}'$ with acceleration factor smaller than k are allocated on CPUs.

Therefore, combining above results before and after t, the assignment $\mathcal{S}$ beginning like HeteroPrio (until time t) and ending like $AreaBound(\mathcal{I}'(t))$ satisfies the following property: $\exists k > 0$ such that all tasks of $\mathcal{I}$ with acceleration factor larger than k are allocated on GPUs and all tasks of $\mathcal{I}$ with acceleration factor smaller than k are allocated on CPUs. This assignment $\mathcal{S}$, whose completion time on both CPUs and GPUs (thanks to Lemma 5.1) is $t + AreaBound(\mathcal{I}')$ clearly defines a solution of the fractional linear program defining the area bound solution, so that $t+AreaBound(\mathcal{I}') \geq AreaBound(\mathcal{I})$.

Similarly, $AreaBound(\mathcal{I})$ satisfies both Lemma 5.2 with some value k' and Lemma 5.1 so that in $AreaBound(\mathcal{I})$, both CPUs and GPUs complete their work simultaneously. If $k' < k$, more work is assigned to GPUs in $AreaBound(\mathcal{I})$ than in $\mathcal{S}$, so that, by considering the completion time on GPUs, we get $AreaBound(\mathcal{I}) \geq t + AreaBound(\mathcal{I}')$. Similarly, if $k' > k$, by considering the completion time on CPUs, we get $AreaBound(\mathcal{I}) \geq t + AreaBound(\mathcal{I}')$. This achieves the proof of Lemma 5.3.

□

Since $AreaBound(\mathcal{I})$ is a lower bound on $C^{Opt}_{\max}(\mathcal{I})$, the above lemma implies that

1. at any time $t \leq T_{FirstIdle}$ in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$, $t + AreaBound(\mathcal{I}'(t)) \leq C^{Opt}_{\max}(\mathcal{I})$,
2. $T_{FirstIdle} \leq C^{Opt}_{\max}(\mathcal{I})$, and thus all tasks start before $C^{Opt}_{\max}(\mathcal{I})$ in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$,
3. if $\forall i \in \mathcal{I}, \max(p_i, q_i) \leq C^{Opt}_{\max}(\mathcal{I})$, then $C^{HP}_{\max}(\mathcal{I}) \leq 2C^{Opt}_{\max}(\mathcal{I})$.

Another interesting characteristic of HeteroPrio is that spoliation can only take place from one type of resource to the other. Indeed, since assignment in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$ is based on the acceleration factors of the tasks, and since a task can only be spoliated if it can be processed faster on the other resource, we get the following lemmas.

Lemma 5.4. *If, in $\mathcal{S}^{NS}_{HP}$, a resource r processes a task whose execution time is not larger on the other resource r', then no task is spoliated from resource r'.*

Proof. Without loss of generality let us assume that there exists a task T executed on a CPU in $\mathcal{S}^{\mathrm{NS}}_{\mathrm{HP}}$, such that $p_T \geq q_T$. We prove that in that case, there is no spoliated task on CPUs, which is the same thing as there being no aborted task on GPUs.

T is executed on a CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, and $\frac{p_T}{q_T} \geq 1$, therefore from HeteroPrio principle, all tasks on GPUs in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ have an acceleration factor at least $\frac{p_{T'}}{q_{T'}} \geq 1$. Non spoliated tasks running on GPUs after $T_{\text{F}irstIdle}$ are candidates to be spoliated by the CPUs. But for each of these tasks, the execution time on CPU is at least as large as the execution time on GPU. It is thus not possible for an idle CPU to spoliate any task running on GPUs, because this task would not complete earlier on the CPU. □

Lemma 5.5. *In HeteroPrio, if a resource executes a spoliated task then no task is spoliated from this resource.*

Proof. Without loss of generality let us assume that T is a spoliated task executed on a CPU. From the HeteroPrio definition, $p_T < q_T$. It also indicates that T was executed on a GPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ with $q_T \geq p_T$. By Lemma 5.4, CPUs do not have any aborted task due to spoliation. □

Finally, we will also rely on the following lemma, that gives the worst case performance of a list schedule when all tasks lengths are large (*i.e.* $\geq C_{\max}^{\text{O}pt}$) on one type of resource.

Lemma 5.6. *Let $\mathcal{B}$ denote a subset of instance $\mathcal{I}$ such that the execution time of each task of $\mathcal{B}$ on one resource is larger than $C_{\max}^{\text{O}pt}(\mathcal{I})$, then any list schedule of $\mathcal{B}$ on $k \geq 1$ resources of the other type has length at most $(2 - \frac{1}{k})C_{\max}^{\text{O}pt}(\mathcal{I})$.*

Proof. Without loss of generality, let us assume that the processing time of each task of set $\mathcal{B}$ on CPU is larger than $C_{\max}^{\text{O}pt}(\mathcal{I})$. All these tasks must therefore be processed on the GPUs in an optimal solution. If scheduling this set $\mathcal{B}$ on k GPUs can be done in time C, then $C \leq C_{\max}^{\text{O}pt}(\mathcal{I})$. The standard list scheduling result from Graham implies that the length of any list schedule of the tasks of $\mathcal{B}$ on GPUs is at most $(2 - \frac{1}{k})C \leq (2 - \frac{1}{k})C_{\max}^{\text{O}pt}(\mathcal{I})$. □

5.5.2 Approximation Ratio with 1 GPU and 1 CPU

Thanks to the above lemmas, we are able to prove an approximation ratio of $\phi = \frac{1+\sqrt{5}}{2}$ for HeteroPrio when the node is composed of 1 CPU and 1 GPU. We will also prove that this result is the best achievable by providing a task set $\mathcal{I}$ for which the approximation ratio of HeteroPrio is ϕ.

Theorem 5.7. *For any instance $\mathcal{I}$ with 1 CPU and 1 GPU, $C_{\max}^{\text{H}P}(\mathcal{I}) \leq \phi C_{\max}^{\text{O}pt}(\mathcal{I})$.*

Proof. Without loss of generality, let us assume that the first idle time (at instant $T_{\text{F}irstIdle}$) occurs on the GPU and the CPU is processing the last remaining task T. We will consider two main cases, depending on the relative values of $T_{\text{F}irstIdle}$ and $(\phi - 1)C_{\max}^{\text{O}pt}$.

1. $T_{\mathrm{FirstIdle}} \leq (\phi - 1)C_{\max}^{Opt}$.
   In $\mathcal{S}_{\mathrm{HP}}^{\mathrm{NS}}$, the finish time of task T is at most $T_{\mathrm{FirstIdle}} + p_T$. If task T is spoliated by the GPU, its execution time is $T_{\mathrm{FirstIdle}} + q_T$. In both cases, the finish time of task T is at most $T_{\mathrm{FirstIdle}} + \min(p_T, q_T) \leq (\phi - 1)C_{\max}^{Opt} + C_{\max}^{Opt} = \phi C_{\max}^{Opt}$.

2. $T_{\mathrm{FirstIdle}} > (\phi - 1)C_{\max}^{Opt}$.
   If T ends before $\phi C_{\max}^{Opt}$ on the CPU in $\mathcal{S}_{\mathrm{HP}}^{\mathrm{NS}}$, since spoliation can only improve the completion time, this ends the proof of the theorem. In what follows, we assume that the completion time of T on the CPU in $\mathcal{S}_{\mathrm{HP}}^{\mathrm{NS}}$ is larger than $\phi C_{\max}^{Opt}(\mathcal{I})$, as depicted in Figure 5.2.

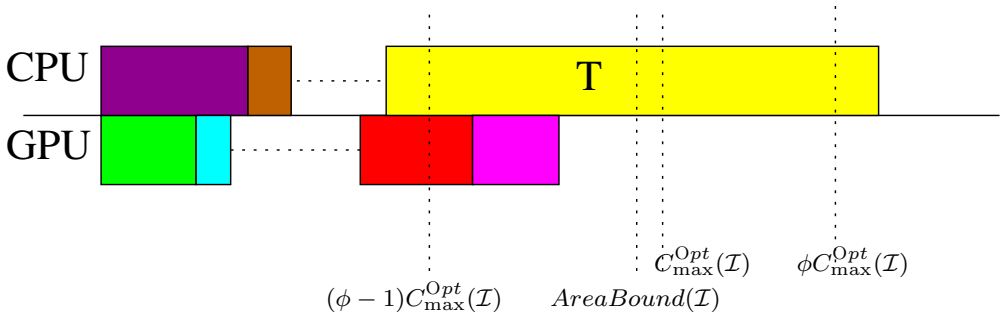


Figure 5.2: Situation where T ends on CPU after $\phi C_{\max}^{Opt}(\mathcal{I})$.

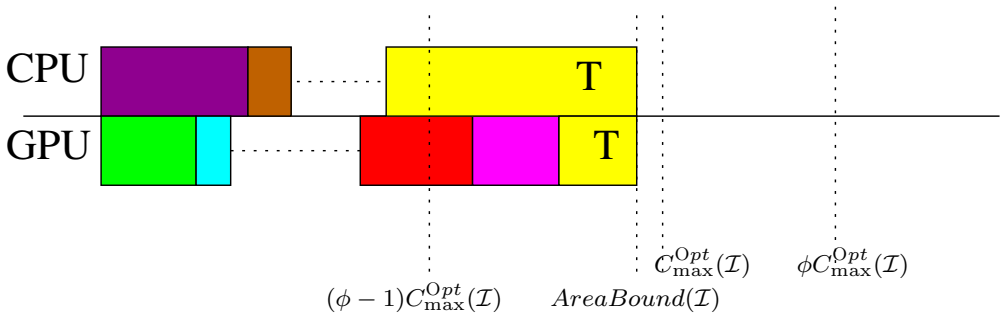


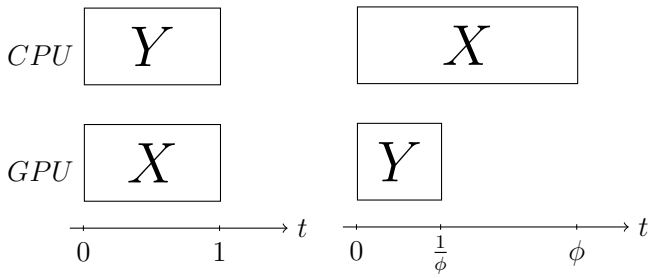
Figure 5.3: Area bound consideration to bound the acceleration factor of T.

It is clear that T is the only unfinished task after $C_{\max}^{Opt}$. Let us denote by α the fraction of T processed after $C_{\max}^{Opt}$ on the CPU. Then $\alpha p_T > (\phi - 1)C_{\max}^{Opt}$ since T ends after $\phi C_{\max}^{Opt}$ by assumption. Lemma 5.3 applied at instant $t = T_{\mathrm{FirstIdle}}$ implies that the GPU is able to process the fraction α of T by $C_{\max}^{Opt}$ (see Figure 5.3) while starting this fraction at $T_{\mathrm{FirstIdle}} \geq (\phi - 1)C_{\max}^{Opt}$ so that $\alpha q_T \leq (1 - (\phi - 1))C_{\max}^{Opt} = (2 - \phi)C_{\max}^{Opt}$. Therefore, the acceleration factor of T is at least $\frac{\phi - 1}{2 - \phi} = \phi$. Since HeteroPrio assigns tasks on the GPU based on their acceleration factors, all tasks in $\mathcal{S}$ assigned to the GPU also have an acceleration factor at least ϕ.

Let us now prove that the GPU is able to process $\mathcal{S} \bigcup \{T\}$ in time $\phi C_{\max}^{Opt}$. Let us split $\mathcal{S} \bigcup \{T\}$ into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ depending on whether the tasks of $\mathcal{S} \bigcup \{T\}$ are processed on the GPU ($\mathcal{S}_1$) or on the CPU ($\mathcal{S}_2$) in the optimal solution. By construction, the processing time of $\mathcal{S}_1$ on the GPU is at most $C_{\max}^{Opt}$ and the processing of $\mathcal{S}_2$ on the CPU takes at most $C_{\max}^{Opt}$. Since the acceleration factor of tasks of $\mathcal{S}_2$ is larger than ϕ, then the processing time of tasks of $\mathcal{S}_2$ on the GPU is at most $C_{\max}^{Opt}/\phi$ and the overall execution of $\mathcal{S} \bigcup \{T\}$ takes at most $C_{\max}^{Opt} + C_{\max}^{Opt}/\phi = \phi C_{\max}^{Opt}$, what ends the proof of the theorem.

□

Theorem 5.8. *The bound of Theorem 5.7 is tight, i.e. there exists an instance $\mathcal{I}$ with 1 CPU and 1 GPU for which HeteroPrio achieves a ratio of ϕ with respect to the optimal solution.*



(a) Optimal schedule (b) HeteroPrio schedule

Figure 5.4: Optimal and HeteroPrio schedules on 1 CPU and 1 GPU.

Proof. Let us consider the instance $\mathcal{I}$ consisting of 2 tasks X and Y, with $p_X = \phi$, $q_X = 1$, $p_Y = 1$ and $q_Y = \frac{1}{\phi}$, such that $\rho_X = \rho_Y = \phi$.

The minimum length of task X is 1, so that $C_{\max}^{Opt} \geq 1$. Moreover, allocating X on the GPU and Y on the CPU leads to a makespan of 1, so that $C_{\max}^{Opt} \leq 1$ and finally $C_{\max}^{Opt} = 1$.

On the other hand, consider the following valid HeteroPrio schedule. The CPU first selects task X and the GPU first selects task Y. The GPU becomes available at instant $\frac{1}{\phi} = \phi - 1$ but does not spoliate task X because it cannot complete X earlier than its expected completion time on the CPU. Therefore, the completion time of HeteroPrio is $\phi = \phi C_{\max}^{Opt}$. □

5.5.3 Approximation Ratio with 1 GPU and m CPUs

In the case of a single GPU and m CPUs, the approximation ratio of HeteroPrio becomes $1 + \phi = \frac{3+\sqrt{5}}{2}$, as proved in Theorem 5.9 and this bound is tight (asymptotically when m becomes large) as proved in Theorem 5.11.

Theorem 5.9. *HeteroPrio achieves an approximation ratio of* $(1+\phi) = \frac{3+\sqrt{5}}{2}$ *for any instance* $\mathcal{I}$ *on* m *CPUs and 1 GPU.*

Proof. Let us assume by contradiction that there exists a task T whose completion time is larger than $(1+\phi)C_{\max}^{Opt}$. We know that all tasks start before $C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. If T is executed on the GPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, then $q_T > C_{\max}^{Opt}$ and thus $p_T \leq C_{\max}^{Opt}$. Since at least one CPU is idle at time $T_{FirstIdle}$, T should have been spoliated and processed by $2C_{\max}^{Opt}$.

We know that T is processed on a CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, and finishes later than $(1+\phi)C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}$. Let us denote by $\mathcal{S}$ the set of all tasks spoliated by the GPU (from a CPU to the GPU) before considering T for spoliation in the execution of HeteroPrio and let us denote by $\mathcal{S}' = \mathcal{S} \bigcup \{T\}$. The following lemma will be used to complete the proof.

Lemma 5.10. *The following holds true*

1. $p_i > C_{\max}^{Opt}$ *for all tasks* i *of* $\mathcal{S}'$,
2. *the acceleration factor of* T *is at least* ϕ,
3. *the acceleration factor of tasks running on the GPU in* $\mathcal{S}_{HP}^{NS}$ *is at least* ϕ.

Proof. of Lemma 5.10. Since all tasks start before $T_{FirstIdle} \leq C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, and since T finishes after $(1+\phi)C_{\max}^{Opt}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, then $p_T > \phi C_{\max}^{Opt}$. Since HeteroPrio performs spoliation of tasks in decreasing order of their completion time, the same applies to all tasks of $\mathcal{S}'$: $\forall i \in \mathcal{S}', p_i > \phi C_{\max}^{Opt}$, and thus $q_i \leq C_{\max}^{Opt}$.
Since $p_T > \phi C_{\max}^{Opt}$ and $q_T \leq C_{\max}^{Opt}$, then $\rho_T > \phi$. Since T is executed on a CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, all tasks executed on GPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ have an acceleration factor at least ϕ. □

Since T is processed on the CPU in $\mathcal{S}_{\text{HP}}^{\text{NS}}$ and $p_T > q_T$, Lemma 5.4 applies and no task is spoliated from the GPU. Let A be the set of tasks running on GPU right after $T_{FirstIdle}$ in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. We consider only one GPU, therefore $|A| \leq 1$.

1. If $A = \{a\}$ with $q_a \leq (\phi-1)C_{\max}^{Opt}$, then Lemma 5.6 applies to $\mathcal{S}'$ (with $n=1$) and the overall completion time is bounded by $T_{FirstIdle} + q_A + C_{\max}^{Opt} \leq (\phi+1)C_{\max}^{Opt}$.
2. If $A = \{a\}$ with $q_a > (\phi-1)C_{\max}^{Opt}$, since $\rho_a > \phi$ by Lemma 5.10, $p_a > \phi(\phi-1)C_{\max}^{Opt} = C_{\max}^{Opt}$. Lemma 5.6 applies to $\mathcal{S}' \bigcup A$, so that the overall completion time is bounded by $T_{FirstIdle} + C_{\max}^{Opt} \leq 2C_{\max}^{Opt}$.
3. If $A = \emptyset$, Lemma 5.6 applies to $\mathcal{S}'$ and get $C_{\max}^{HP}(\mathcal{I}) \leq T_{FirstIdle} + C_{\max}^{Opt} \leq 2C_{\max}^{Opt}$.

Therefore, in all cases, the completion time of task T is at most $(\phi+1)C_{\max}^{Opt}$, what achieves the proof of Theorem 5.9. □

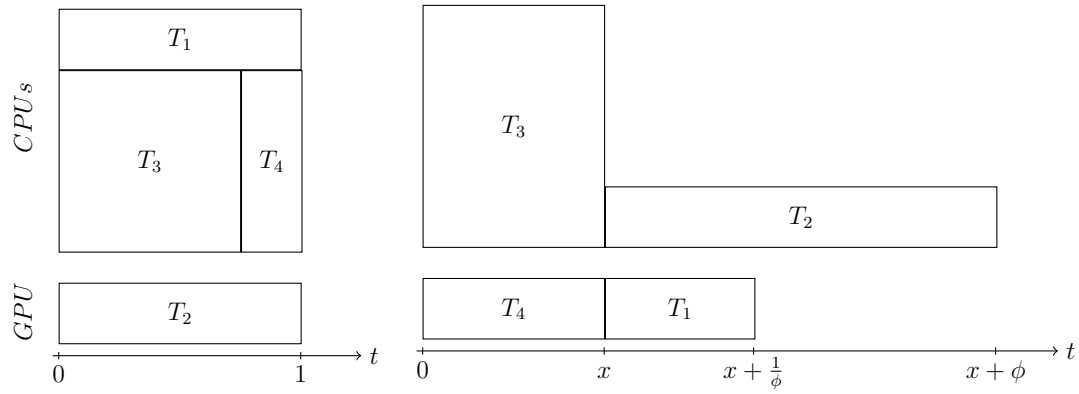
Theorem 5.11. *Theorem 5.9 is tight, i.e. for any $\delta > 0$, there exists an instance $\mathcal{I}$ such that $C_{\max}^{HP}(\mathcal{I}) \geq (\phi + 1 - \delta) C_{\max}^{Opt}(\mathcal{I})$.*

Proof. For some $\epsilon > 0$, let $\mathcal{I}$ denote the following set of tasks:

Task	CPU Time	GPU Time	# of tasks	accel ratio
T_1	1	$1/\phi$	1	ϕ
T_2	ϕ	1	1	ϕ
T_3	ϵ	ϵ	$(mx)/\epsilon$	1
T_4	$\epsilon\phi$	ϵ	x/ϵ	ϕ

where $x = \frac{m-1}{m+\phi}$.

The minimum length of task T_2 is 1, so that $C_{\max}^{Opt} \geq 1$. Moreover, if T_1, T_3 and T_4 are scheduled on CPUs and T_2 on the GPU (this is possible if ϵ is small enough), then the completion time is 1, so that $C_{\max}^{Opt} = 1$.



(a) Optimal schedule (b) HeteroPrio schedule

Figure 5.5: Optimal and HeteroPrio schedules on m CPUs and 1 GPU.

Consider the following valid HeteroPrio schedule. The GPU first selects tasks from T_4 and the CPUs first select tasks from T_3. All resources become available at time x. Now, the GPU selects task T_1 and one of the CPUs selects task T_2, with a completion time of $x+\phi$. The GPU becomes available at $x+1/\phi$ but does not spoliate T_2 since it would not finish before $x + 1/\phi + 1 = x + \phi$. The makespan of HeteroPrio is thus $x + \phi$, and since x tends towards 1 when m becomes large, the approximation ratio of HeteroPrio on this instance tends towards $1 + \phi$. □

5.5.4 Approximation Ratio with n GPUs and m CPUs

In the most general case of n GPUs and m CPUs, the approximation ratio of HeteroPrio is at most $2 + \sqrt{2}$, as proved in Theorem 5.12. To establish this result, we rely on the same techniques as in the case of a single GPU,

but the result of Lemma 5.6 is weaker for $n > 1$, what explains that the approximation ratio is larger than for Theorem 5.9. We have not been able to prove, as previously, that this bound is tight, but we provide in Theorem 5.17 a family of instances for which the approximation ratio is arbitrarily close to $2 + \frac{2}{\sqrt{3}}$.

Theorem 5.12. *For any instance $\mathcal{I}$, $C^{HP}_{\max}(\mathcal{I}) \leq (2+\sqrt{2})C^{Opt}_{\max}(\mathcal{I})$.*

Proof. We prove this by contradiction. Let us assume that there exists a task T whose completion time in $\mathcal{S}_{\text{HP}}$ is larger than $(2+\sqrt{2})C^{Opt}_{\max}$. Without loss of generality, we assume that T is executed on a CPU in $\mathcal{S}^{\text{NS}}_{\text{HP}}$. In the rest of the proof, we denote by $\mathcal{S}$ the set of all tasks spoliated by GPUs in the HeteroPrio solution, and $\mathcal{S}' = \mathcal{S} \cup \{T\}$. The following lemma will be used to complete the proof.

Lemma 5.13. *The following holds true.*

1. $\forall i \in \mathcal{S}', p_i > C^{Opt}_{\max}$
2. *All tasks T' executed on a GPU in $\mathcal{S}^{NS}_{HP}$ have $\rho_{T'} \geq 1+\sqrt{2}$.*

Proof. of Lemma 5.13. In $\mathcal{S}^{\text{NS}}_{\text{HP}}$, all tasks start before $T_{\text{F}irstIdle} \leq C^{Opt}_{\max}$. Since T ends after $(2+\sqrt{2})C^{Opt}_{\max}$ in $\mathcal{S}^{\text{NS}}_{\text{HP}}$ (since spoliation can only improve the completion time), then $p_T > (1+\sqrt{2})C^{Opt}_{\max}$. The same applies to all spoliated tasks that complete after T in $\mathcal{S}^{\text{NS}}_{\text{HP}}$. If T is not considered for spoliation, no task that complete before T in $\mathcal{S}^{\text{NS}}_{\text{HP}}$ is spoliated, and the first result holds. Otherwise, let s_T denote the instant at which T is considered for spoliation. The completion time of T in $\mathcal{S}_{\text{HP}}$ is at most $s_T + q_T$, and since $q_T \leq C^{Opt}_{\max}$, $s_T \geq (1+\sqrt{2})C^{Opt}_{\max}$. Since HeteroPrio handles tasks for spoliation in decreasing order of their completion time in $\mathcal{S}^{\text{NS}}_{\text{HP}}$, tasks T' is spoliated after T has been considered and not finished at time s_T, and thus $p_{T'} > \sqrt{2}C^{Opt}_{\max}$.
Since $p_T > (1+\sqrt{2})C^{Opt}_{\max}$ and $q_T \leq C^{Opt}_{\max}$, then $\rho_T \geq (1+\sqrt{2})$. Since T is executed on a CPU in $\mathcal{S}^{\text{NS}}_{\text{HP}}$, all tasks executed on GPU in $\mathcal{S}^{\text{NS}}_{\text{HP}}$ have acceleration factor at least $1+\sqrt{2}$. □

Let $\mathcal{A}$ be the set of tasks executed on GPUs after time $T_{\text{F}irstIdle}$ in $\mathcal{S}^{\text{NS}}_{\text{HP}}$. We partition $\mathcal{A}$ into two sets $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\forall i \in \mathcal{A}_1, q_i \leq \frac{C^{Opt}_{\max}}{\sqrt{2}+1}$ and $\forall i \in \mathcal{A}_2, q_i > \frac{C^{Opt}_{\max}}{\sqrt{2}+1}$.

Since there are n GPUs, $|\mathcal{A}_1| \leq |\mathcal{A}| \leq n$. We consider the schedule induced by HeteroPrio on the GPUs with the tasks $\mathcal{A} \bigcup \mathcal{S}'$ (if T is spoliated, this schedule is actually returned by HeteroPrio, otherwise this is what HeteroPrio builds when attempt to spoliate task T). This schedule is not worse than a schedule that processes all tasks from $\mathcal{A}_1$ starting at time $T_{\text{F}irstIdle}$, and then performs any list schedule of all tasks from $\mathcal{A}_2 \bigcup \mathcal{S}'$. Since $|\mathcal{A}_1| \leq n$, the first part takes time at most $\frac{C^{Opt}_{\max}}{\sqrt{2}+1}$. For all T_i in $\mathcal{A}_2$, $\rho_i \geq 1+\sqrt{2}$ and $q_i > \frac{C^{Opt}_{\max}(\mathcal{I})}{\sqrt{2}+1}$

imply $p_i > C_{\max}^{Opt}$. We can thus apply Lemma 5.6 to $\mathcal{A}_2 \bigcup \mathcal{S}'$ and the second part takes time at most $2C_{\max}^{Opt}$. Overall, the completion time on GPUs is bounded by $T_{FirstIdle} + \frac{C_{\max}^{Opt}}{\sqrt{2}+1} + (2-\frac{1}{n})C_{\max}^{Opt} < C_{\max}^{Opt} + (\sqrt{2}-1)C_{\max}^{Opt} + 2C_{\max}^{Opt} = (\sqrt{2}+2)C_{\max}^{Opt}$, which is a contradiction.

□

Now we provide different instances to show the worst case behavior of HeteroPrio.

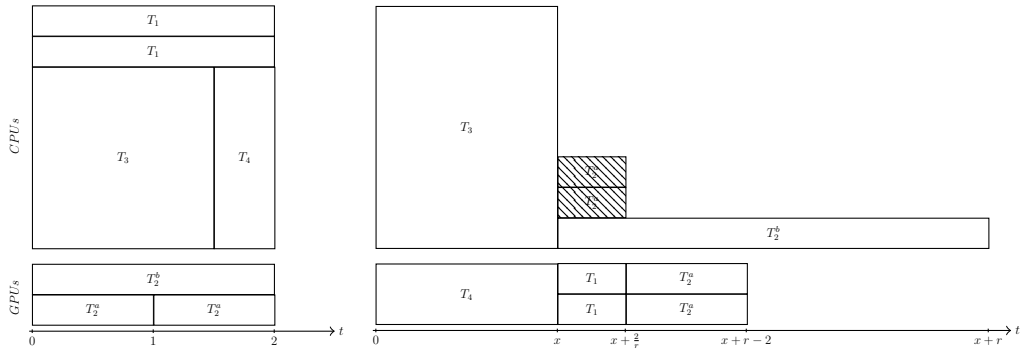
Theorem 5.14. *On a system consisting of m CPUs and 2 GPUs, HeteroPrio can achieve a ratio as large as* $\simeq 2.78$ *with respect to optimal completion time.*

Proof. Let $\mathcal{I}$ denote the following set of tasks:

Task	CPU Time	GPU Time	# of tasks	accel ratio
T_1	2	$\frac{2}{r}$	2	r
T_2^a	r	1	2	r
T_2^b	r	2	1	$\frac{r}{2}$
T_3	ϵ	$\frac{\epsilon}{a}$	$\frac{mx}{\epsilon}$	a
T_4	ϵr	ϵ	$\frac{2x}{\epsilon}$	r

where, $1 \leq a \leq \frac{r}{2}$, $x = \frac{m-2}{m+2r}2$, $m >> 2$ and $r = \frac{3+\sqrt{17}}{2}$ is the solution of the equation $\frac{2}{r} + 3 = r$.

Minimum length of task T_2^b is 2, so that $C_{\max}^{Opt} \geq 2$. Moreover, if tasks T_1, T_3 and T_4 are scheduled on the CPUs (the overall work is $4 + mx + 2xr = 2m$ and we assume ϵ is chosen such that tasks can be packed together) and if T_2^* are processed on the GPUs (1 with the 2 T_2^a tasks and 1 with T_2^b task), the overall completion time is 2, so that $C_{\max}^{Opt} \leq 2$. Then, overall $C_{\max}^{Opt} = 2$.



(a) Optimal schedule (b) HeteroPrio schedule

Figure 5.6: Optimal and HeteroPrio schedules on m CPUs and 2 GPUs. Aborted tasks in HeteroPrio are shown in pattern boxes.

Let us now consider the following valid HeteroPrio schedule. GPUs first select T_4 tasks (with maximal acceleration factor r) and CPUs first select T_3

tasks (with minimal acceleration factor $a \leq \frac{r}{2}$). Both CPUs and GPUs become available at time x.
Now GPUs select T_1 tasks (with maximal acceleration factor r) and 3 CPUs (since there are only 3 available tasks) select T_2^* tasks (with acceleration factor $\leq r$). GPUs end up first and complete their work at instant $x+2/r = r-3+x$. Since the completion time of T_2^* tasks on the CPUs is $r+x$, then 2 tasks are spoliated by GPUs (both T_2^a tasks) and complete on the GPUs at instant $r-2+x$.
Then, at instant $r-2+x$ the spoliation of the last T_2^b task would not improve its completion time and the overall completion time is $r+x$.

Therefore, the overall completion time of HeteroPrio is $r+x = \frac{m-2}{m+2r}C_{\max}^{Opt}+r$ that becomes arbitrarily close to $\frac{7+\sqrt{17}}{2} \simeq 2.78C_{\max}^{Opt}$ when m becomes large. □

Remark 1. *We can easily extend above example to 3 GPUs using the following set of tasks I, where HeteroPrio can achieve a ratio as large as* $\simeq 2.84$ *with respect to optimal completion time.*

TaskName	*CPU Time*	*GPU Time*	*# of tasks*	*accel ratios*
T_1	3	$\frac{3}{r}$	*3*	r
T_2^a	r	1	2	r
T_2^b	r	2	2	$\frac{r}{2}$
T_2^c	r	3	1	$\frac{r}{3}$
T_3	ϵ	$\frac{\epsilon}{a}$	$\frac{mx}{\epsilon}$	a
T_4	ϵr	ϵ	$\frac{3x}{\epsilon}$	r

where, $1 \leq a \leq \frac{r}{3}$, $x = \frac{m-3}{m+3r}3$, $m >> 3$ *and* $r = \frac{5+\sqrt{37}}{2}$ *is the solution of the equation* $\frac{3}{r}+5 = r$.

Remark 2. *It is even possible to extend above example to n GPUs and to achieve a ratio of* $\frac{3n-1}{n} + \frac{2}{2n-1+\sqrt{(2n-1)^2+4n}}$ *with respect to optimal completion time (above cases correspond to* $n = 2$ *and* $n = 3$ *respectively). When n becomes large,* $\frac{3n-1}{n} + \frac{2}{2n-1+\sqrt{(2n-1)^2+4n}} = 3 - \frac{1}{n} + o(\frac{1}{n})$*, so that HeteroPrio can achieve an approximation ratio as bad as 3 when n (and m) becomes large.*

In all previous instances the smallest task of set T_2 (tasks processed on GPUs in optimal solution) is of unit length. Now we provide a family of instances where the smallest task of set T_2 is one third of optimal makespan. It allows tasks of set T_2 to have a large execution time on CPU without having large acceleration factor.

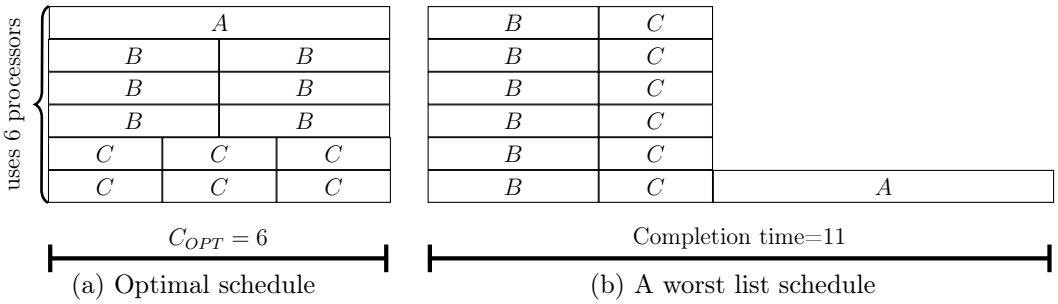
Theorem 5.15. *On 6 homogeneous processors, it is possible to construct a* $2-\frac{1}{6}$ *times worse list schedule for an instance* $\mathcal{I}$*, whose all tasks are atleast of length* $\frac{C_{\max}^{Opt}(\mathcal{I})}{3}$.

Proof. Makespan of worst list schedule is almost twice the optimal makespan is a well known result. But here we exhibit that length of the smallest task is as large as one third of the optimal makespan. Let us consider $\mathcal{I}$ be the following set of tasks which we want to schedule on 6 homogeneous processors.

TaskName	length	# of tasks
A	6	1
B	3	6
C	2	6

Since the length of task A is 6, therefore $C_{\max}^{Opt} \geq 6$. Figure 5.7a exhibits a schedule whose overall makespan is 6, therefore $C_{\max}^{Opt} = 6$.

Figure 5.7b exhibits a schedule of length 11, which is $2 - \frac{1}{6}$ times of $C_{\max}^{Opt}$. The smallest task of the considered set, C, is of length $\frac{C_{\max}^{Opt}}{3} = 2$. □



(a) Optimal schedule (b) A worst list schedule

Figure 5.7: Schedules on 6 homogeneous processors.

Theorem 5.16. *On a system consisting of m CPUs and 6 GPUs, HeteroPrio can achieve a ratio as large as* 3 *with respect to optimal completion time.*

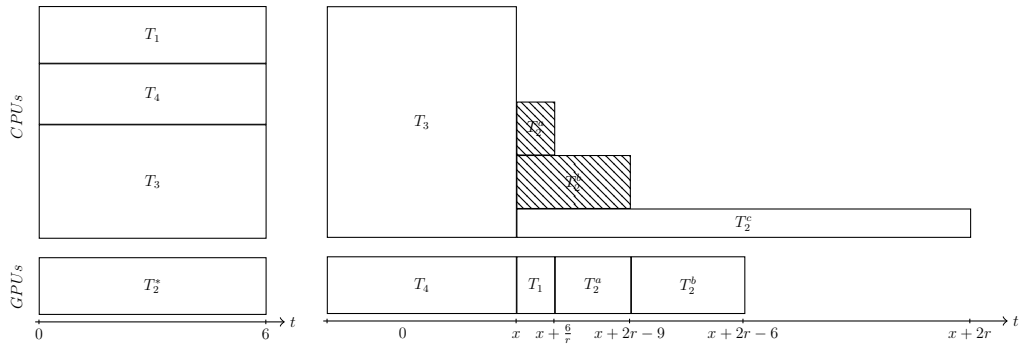
Proof. Let $\mathcal{I}$ denote the following set of tasks:

Task	CPU Time	GPU Time	# of tasks	accel ratio
T_1	6	$\frac{6}{r}$	6	r
T_2^a	$2r$	2	6	r
T_2^b	$2r$	3	6	$\frac{2r}{3}$
T_2^c	$2r$	6	1	$\frac{r}{3}$
T_3	ϵ	$\frac{\epsilon}{a}$	$\frac{mx}{\epsilon}$	a
T_4	ϵr	ϵ	$\frac{6x}{\epsilon}$	r

where, $1 \leq a \leq r\frac{min\{2,3,6\}}{max\{2,3,6\}} = \frac{r}{3}$, $x = \frac{m-6}{m+6r}6$, $m >> 6$ and $r = 6$ is the solution of the equation $\frac{6}{r} + 11 = 2r$.

Minimum length of task T_2^c is 6, so that $C_{\max}^{Opt} \geq 6$. Moreover, if tasks T_1, T_3 and T_4 are scheduled on the CPUs (the overall work is $36 + mx + 6xr = 6m$ and we assume ϵ is chosen such that tasks can be packed together) and if T_2^*

are processed on the GPUs (similar to as shown in Figure 5.7a), the overall completion time is 6, so that $C_{\max}^{Opt} \leq 6$. Then, overall $C_{\max}^{Opt} = 6$.



(a) Optimal schedule (b) HeteroPrio schedule

Figure 5.8: Optimal and HeteroPrio schedules on m CPUs and 6 GPUs. Aborted tasks in HeteroPrio are shown in pattern boxes.

Let us now consider the following valid HeteroPrio schedule. GPUs first select T_4 tasks (with maximal acceleration factor r) and CPUs first select T_3 tasks (with minimal acceleration factor $a \leq \frac{r}{3}$). Then, all CPUs and GPUs become available at time x.
Then, each GPU first selects a T_1 task (with maximal acceleration factor r) and 13 CPUs (since there are only 13 available tasks) select T_2^* tasks (with acceleration factor $\leq r$). Then, GPUs end up first and complete their work at instant $x + 6/r = 2r - 11 + x$.
Since the completion time of T_2^* tasks on the CPUs is $2r + x$, then 6 tasks are spoliated by GPUs (all T_2^a tasks) and complete on the GPUs at instant $2r - 9 + x$. Also all T_2^b tasks are spoliated by GPUs at instant $2r - 9 + x$ and complete on GPUs at $2r - 6 + x$
Then, at instant $2r-6+x$ the spoliation of the last T_2^c task would not improve its completion time and the overall completion time is $2r + x$.

Therefore, the overall completion time is $2r + x = \frac{(m-6)}{m+6r} C_{\max}^{Opt} + 2r$ that becomes arbitrarily close to $18 = 3C_{\max}^{Opt}$ when m becomes large.

□

We extend above example to n GPUs and provide the lower bound for HeteroPrio in Theorem 5.17.

Theorem 5.17. *The approximation ratio of HeteroPrio is at least* $2 + \frac{2}{\sqrt{3}} \simeq 3.15$.

Proof. We consider an instance $\mathcal{I}$, with $n = 6k$ GPUs and $m = n^2$ CPUs, containing the following tasks.

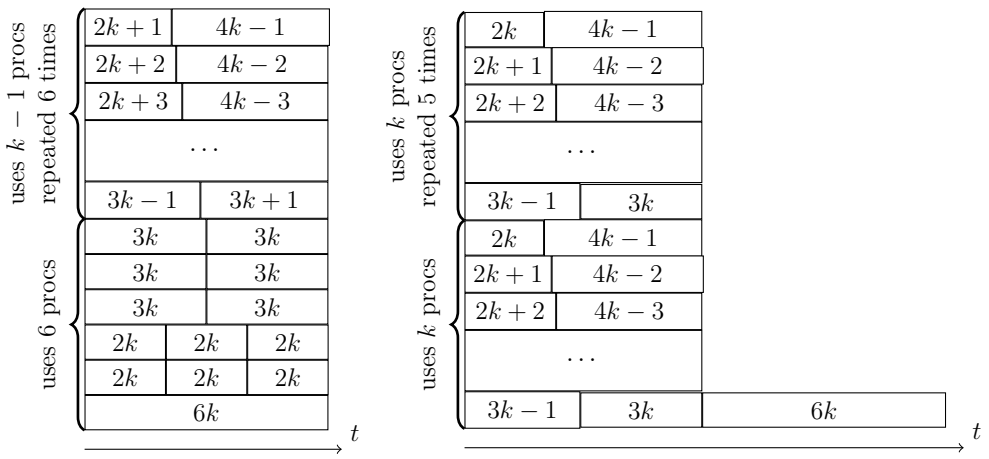


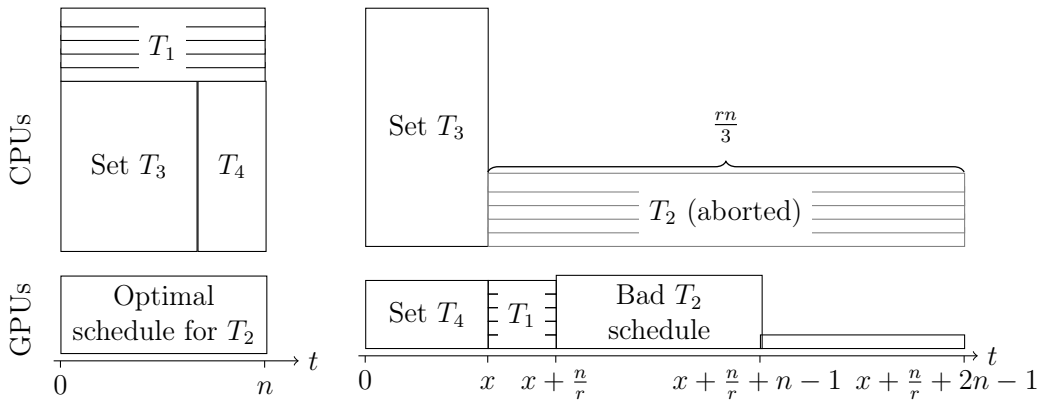
Figure 5.9: Two schedules for task set T_2 on $n = 6k$ homogeneous processors. Tasks are labeled with their processing times. Left one is an optimal schedule and right one is a possible list schedule.

Task	CPU Time	GPU Time	# of tasks	accel ratio
T_1	n	$\frac{n}{r}$	n	r
T_2	$\frac{rn}{3}$	see below	see below	$\frac{r}{3} \leq \rho \leq r$
T_3	1	1	mx	1
T_4	r	1	nx	r

,

where $x = \frac{m-n}{m+nr}n$ and r is the solution of the equation $\frac{n}{r} + 2n - 1 = \frac{nr}{3}$. Note that the highest acceleration factor is r and the lowest is 1 since $r > 3$. The set T_2 contains tasks with the following execution time on GPU,
(i) one task of length $n = 6k$, (ii) for all $0 \leq i \leq 2k-1$, six tasks of length $2k+i$. This set T_2 is a generalization of tasks considered in Theorem 5.15.

Tasks of set T_2 can be scheduled on n GPUs in time n (see Figure 5.9). $\forall 1 \leq i < k$, each of the six tasks of length $2k+i$ can be combined with one of the six tasks of length $2k+(2k-i)$, occupying $6(k-1)$ processors; the tasks of length $3k$ can be combined together on 3 processors, and there remains 3 processors for the six tasks of length $2k$ and the task of length $6k$. On the other hand, a worst list schedule may achieve makespan $2n-1$ on the n GPUs. $\forall 0 \leq i \leq k-1$, each of the six tasks of length $2k+i$ is combined with one of the six tasks of length $4k-i-1$, which occupies all $6k$ processors until time $6k-1$, then the task of length $6k$ is executed. The fact that there exists a set of tasks for which the makespan of a worst case list schedule is almost twice the optimal makespan is a well known result. However, the interest of set T_2 is that the smallest execution time is $C_{\max}^{Opt}(T_2)/3$, what allows these tasks to have a large execution time on CPU in instance $\mathcal{I}$ (without having a too large



(a) Optimal schedule (b) HeteroPrio schedule when trying to spoliate the last task

Figure 5.10: Optimal and HeteroPrio schedules for the instance of Theorem 5.17.

acceleration factor).

Figure 5.10a shows an optimal schedule of length n for this instance: the tasks from set T_2 are scheduled optimally on the n GPUs, and the sets T_1, T_3 and T_4 are scheduled on the CPUs. Tasks T_3 and T_4 fit on the $m - n$ CPUs because the total work is $mx + nxr = x(m + nr) = (m - n)n$ by definition of x.

On the other hand, Figure 5.10b shows a possible HeteroPrio schedule for $\mathcal{I}$. The tasks from set T_3 have the lowest acceleration factor and are scheduled on the CPUs, while tasks from T_4 are scheduled on the GPUs. All resources become available at time x. Tasks from set T_1 are scheduled on the n GPUs, and tasks from set T_2 are scheduled on m CPUs. At time $x + \frac{n}{r}$, the GPUs become available and start spoliating the tasks from set T_2. Since they all complete at the same time, the order in which they get spoliated can be arbitrary, and it can lead to the worst case behavior of Figure 5.9, where the task of length n is executed last. In this case, spoliating this task does not improve its completion time, and the resulting makespan for HeteroPrio on this instance is $C^{HP}_{\max}(I) = x + \frac{n}{r} + 2n - 1 = x + \frac{nr}{3}$ by definition of r.
The approximation ratio on this instance is thus $\frac{C^{HP}_{\max}(I)}{C^{Opt}_{\max}(I)} = \frac{x}{n} + \frac{r}{3}$. When n becomes large, $\frac{x}{n}$ tends towards 1, and r tends towards $3 + 2\sqrt{3}$. Hence, the ratio $\frac{C^{HP}_{\max}(I)}{C^{Opt}_{\max}(I)}$ tends towards $2 + \frac{2}{\sqrt{3}}$, what ends the proof. □

5.6 Experimental evaluation

In this section, we propose another experimental evaluation of HeteroPrio on instances coming from the dense linear algebra library Chameleon [8]. We

evaluate our algorithms in two contexts, (i) with independent tasks and (ii) with dependencies, which is closer to real-life settings and is ultimately the goal of the HeteroPrio algorithm. In this section, we use task graphs from Cholesky, QR and LU factorizations, which provide interesting insights on the behavior of the algorithms. As mentioned in Chapter 1, The Chameleon library is built on top of the StarPU runtime, and implements tiled versions of many linear algebra kernels expressed as graphs of tasks. Before the execution, the processing times of the tasks are measured on both types of resources, which then allows StarPU schedulers to have a reliable prediction of each task's processing time. In this section, we use this data to build input instances for our algorithms, obtained on a machine with 20 CPU cores of two Haswell Intel® Xeon® E5-2680 processors and 4 Nvidia K40-M GPUs. It is the same machine what we used in Chapter 4. We consider Cholesky, QR and LU factorizations with a tile size of 960, and a number of tiles N varying between 4 and 64.

We compare 3 algorithms from the literature : HeteroPrio, the well-known HEFT algorithm (designed for the general $R|prec|C_{\max}$ problem), and DualHP from [29] (specifically designed for CPU and GPU, with an approximation ratio of 2 for independent tasks). The DualHP algorithm works as follows : for a given guess λ on the makespan, it either returns a schedule of length 2λ, or ensures that $\lambda < C_{\max}^{Opt}$. To achieve this, any task with processing time more than λ on any resource is assigned to the other resource, and then all remaining tasks are assigned to the GPU by decreasing acceleration factor while the overall load is lower than $n\lambda$. If the remaining load on CPU is not more than $m\lambda$, the resulting schedule has makespan below 2λ. The best value of λ is then found by binary search.

5.6.1 Independent Tasks

To obtain realistic instances with independent tasks, we have taken the actual measurements from tasks of each kernel (Cholesky, QR and LU) and considered these as independent tasks. For each instance, the performance of all three algorithms is compared to the area bound. Results are depicted in Figure 5.11, where the ratio to the area bound is given for different values of the number of tiles N.

The results show that both HeteroPrio and DualHP achieve close to optimal performance when N is large, but HeteroPrio achieves better results for small values of N (below 20). This may be surprising, since the approximation ratio of DualHP is actually better than the one of HeteroPrio. On the other hand, HeteroPrio is primarily a list scheduling algorithm, that usually achieve good average case performance. In this case, it comes from the fact that DualHP tends to balance the *load* between the set of CPUs and the set of GPUs, but for such small values of N, the task processing times on CPU are not negligible

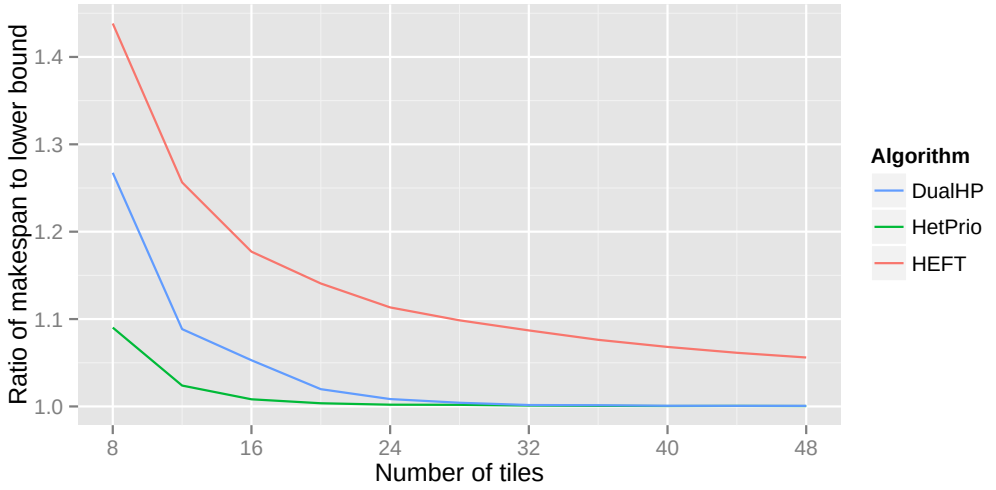


Figure 5.11: Results for independent tasks.

compared to the makespan. Thus, it happens that average loads are similar for both kinds of resources, but one CPU actually has significantly higher load than the others, what results in a larger makespan. HEFT, on the other hand, has rather poor performance because it does not take acceleration factor into account, and thus assigns tasks to GPUs that would be better suited to CPUs, and vice-versa.

5.6.2 Task Graphs

Both HeteroPrio and DualHP can be easily adapted to take dependencies into account, by applying at any instant the algorithm on the current set of ready tasks. For DualHP, this implies recomputing the assignment of tasks to resources each time a task becomes ready, and also slightly modifying the algorithm to take into account the load of currently executing tasks. Since HeteroPrio is a list algorithm, HeteroPrio rule can be used to assign a ready task to any idle resource. If no ready task is available for an idle resource, a spoliation attempt is made on currently running tasks.

When scheduling task graphs, a standard approach is to compute task priorities based on the dependencies. For homogeneous platforms, the most common priority scheme is to compute the *bottom-level* of each task, *i.e.* the maximum length of a path from this task to the exit task, where nodes of the graph are weighted with the execution time of the corresponding task. In the heterogeneous case, the priority scheme used in the standard HEFT algorithm is to set the weight of each node as the average execution time of the corresponding task on all resources. We will denote this scheme by `avg`. A more optimistic view could be to set the weight of each node as the smallest

execution time on all resources, hoping that tasks will get executed on their favorite resources. We will denote this scheme `min`.

In both HeteroPrio and DualHP, these ranking schemes are used to break ties. In HeteroPrio, whenever two tasks have the same acceleration factor, the highest priority task is assigned first; furthermore, when several tasks can be spoliated for some resource, the highest priority candidate is selected. In DualHP, once the assignment of tasks to CPUs and GPUs is computed, tasks are sorted by highest priority first and processed in this order. For DualHP, we also consider another ranking scheme, `fifo`, in which no priority is computed and tasks are assigned in the order in which they became ready.

We thus consider a total of 7 algorithms: HeteroPrio, DualHP and HEFT with `min` and `avg` ranking schemes, and DualHP with `fifo` ranking scheme. We again consider three types of task graphs: Cholesky, QR and LU factorizations, with the number of tiles N varying from 4 to 64. For each task graph, the makespan with each algorithm is computed, and we consider the ratio to the lower bound obtained by adding dependency constraints to the area bound [15]. Results are depicted in Figure 5.12.

The first conclusion from these results is that scheduling DAGs corresponding to small or large values of N is relatively easy, and all algorithms achieve performance close to the lower bound: with small values of N, the makespan is constrained by the critical path of the graph, and executing all tasks on GPU is the best option; when N is large, the available parallelism is large enough, and the runtime is dominated by the available work. The interesting part of the results is thus for the intermediate values of N, between 10 and 30 or 40 depending on the task graph. In these cases, the best results are always achieved by HeteroPrio, especially with the `min` ranking scheme, which is always within 30% of the (optimistic) lower bound. On the other hand, all other algorithms get significantly worse performance for at least one case.

To obtain a better insight on these results, let us further analyze the schedules produced by each algorithm by focusing on the following metrics: the amount of idle time on each type of resources (CPU and GPU)[1], and the adequacy of task allocation (whether the tasks allocated to each resource is a good fit or not). To measure the adequacy of task allocation on a resource r, we define the acceleration factor $\mathcal{A}_r$ of the "equivalent task" made of all the tasks assigned to that resource: let J be the set of tasks assigned to r, $\mathcal{A}_r = \frac{\sum_{i \in J} p_i}{\sum_{i \in J} q_i}$. A schedule has a good adequacy of task allocation if $\mathcal{A}_{\text{GPU}}$ is high and $\mathcal{A}_{\text{CPU}}$ is low. The values of equivalent acceleration factors for both resources are shown on Figure 5.13. On Figure 5.14, the normalized idle time on each resource is depicted, which is the ratio of the idle time on a resource to the amount of that resource used in the lower bound solution.

[1]For fairness, any work made on an "aborted" task by HeteroPrio is also counted as idle time, so that all algorithms have the same amount of work to execute.

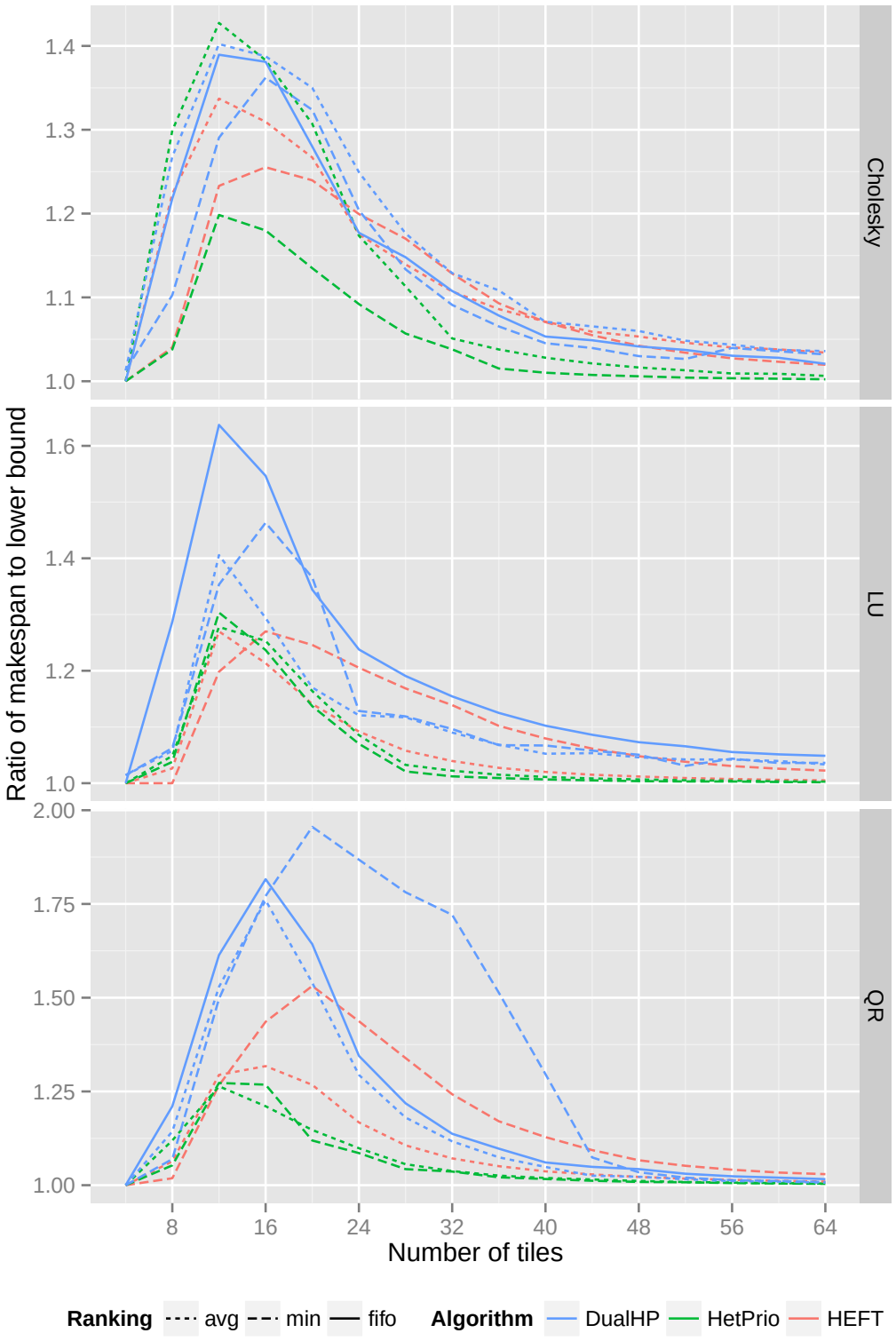


Figure 5.12: Results for different DAGs.

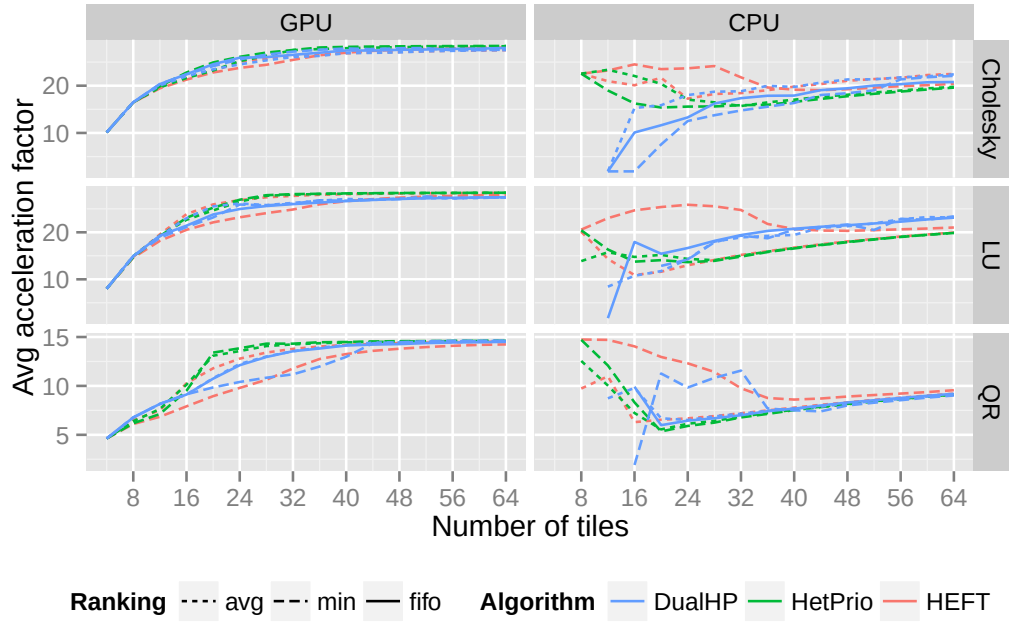


Figure 5.13: Equivalent acceleration factors.

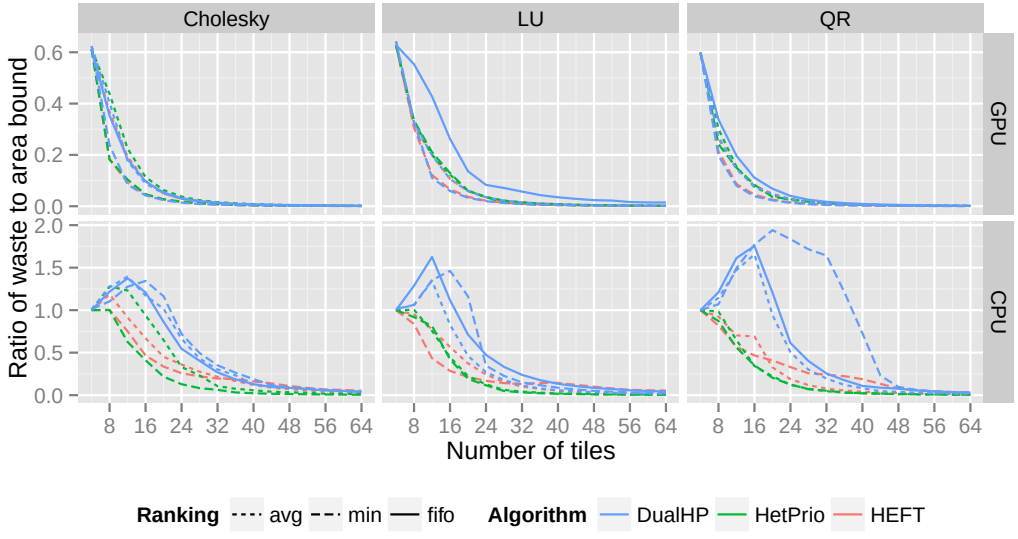


Figure 5.14: Normalized idle time.

On Figure 5.13, one can observe that there are significant differences in the acceleration factor of tasks assigned to the CPU between the different algorithms. In particular, HeteroPrio usually assigns to the CPU tasks with low acceleration factor (which is good), whereas HEFT usually has a higher acceleration factor on CPU. DualHP is somewhat in the middle, with a few exceptions in the case of LU when N is large. On the other hand, Figure 5.14 shows that HEFT and HeteroPrio are able to keep relatively low idle times in all cases, whereas DualHP induces very large idle time on the CPU. The reason for this is that optimizing locally the makespan for the currently available tasks makes the algorithm too conservative, especially at the beginning of the schedule where there are not many ready tasks, DualHP assigns all tasks on the GPU because assigning one on the CPU would induce a larger completion time. HeteroPrio however is able to find a good compromise by keeping the CPU busy with the tasks that are not well suited for the GPU, and relies on the spoliation mechanism to ensure that bad decisions do not penalize the makespan.

5.7 Conclusion

In this chapter, we analyze the theoretical and practical performance of HeteroPrio for scheduling independent tasks on two types of unrelated resources. HeteroPrio has been proposed in a practical context, and we provide theoretical worst-case approximation proofs in several cases, including the most general, and we prove that our bounds are tight.

Furthermore, we show experimentally that with DAGs coming from Linear Algebra, HeteroPrio produces very efficient schedules, whose makespans are better than the state-of-the-art algorithms from the literature, and very close to the theoretical lower bounds. In future, we plan to work on approximation bounds of HeteroPrio and other state-of-the-art scheduling algorithms for general task graphs.

This page is intentionally left blank.

Conclusion

In this thesis, we have developed different scheduling techniques to efficiently exploit the capabilities of modern heterogeneous platforms for task based dense linear algebra applications. We have shown special interest in improving the performance of dense Cholesky factorization on different heterogeneous platforms composed of CPUs and GPUs. We proposed and evaluated different static and dynamic strategies. We have shown that introducing some static information into the dynamic task scheduler improves the performance of an application significantly. We have also shown that static schedules (for Cholesky factorization) are robust to variations in execution timings. We proposed different upper bounds on the performance of task graphs and used them to assess the quality of different schedules.

A resource centric dynamic scheduler, HeteroPrio, has been proposed recently for a set of small independent tasks on two types of resources, which is based on the affinity between tasks and resources. We extended this scheduler and proposed a family of HeteroPrio on two types of resource, for general task graphs, that greatly benefits from basic qualitative information about the task graph. Later, we provided several extensions of the HeteroPrio scheduling strategy to the case with more than two types of resources and evaluated these extensions on a platform composed of a single CPU, cluster of CPUs and GPUs. HeteroPrio is based on affinity between tasks and resources and we do not formulate communication costs explicitly, still in all scenarios (on two types of resources as well as on more than two types of resources), we observe that most of the HeteroPrio variants are better than sate-of-the-art HEFT heuristic for Cholesky factorization of medium size matrices. It indicates that HeteroPrio scheduler is very effective for cases where scheduling decisions are crucial in order to achieve good performance. Lastly, we also provided a theoretical insight on the performance of HeteroPrio by proving several approximation bounds for a set of independent tasks on two types of resources.

We present the detailed contributions of different chapters in the following paragraphs.

In Chapter 2, we proposed improved performance bounds, which take into account both resource and task heterogeneity, as well as critical paths. 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.

In Chapter 3, we provided a fair comparison between static and dynamic scheduling strategies on heterogeneous platforms consisting of CPU and GPU nodes. The development of dynamic schedulers on runtime systems is motivated by expected weaknesses and limitations of static schedulers. It has been observed that execution times of kernels in nodes where many resources (cache, memories, buses) are shared suffer high variance and it is generally assumed that the difficulty to predict execution times makes static schedulers useless. We proved that this assertion is in general not true and that static schedules (for Cholesky factorization) are in fact robust to variations in execution times. We have also proved that combining dynamic strategies with simulation in order to build less myopic algorithms can significantly improve their performance. We also considered a family of dynamic schedulers (HeteroPrio) that performs poorly on general graphs but greatly benefits from basic qualitative information about the task graph.

In Chapter 4, we presented several extensions of the HeteroPrio scheduling strategy to the case with more than two types of resources. Besides the obvious case of platforms with different accelerator types, this capability is also crucial when CPU cores are clustered together to make use of intra-task parallelism, as it has been recently advocated in order to make a better use of all available resources and to build a more homogeneous platform. We exhibited that HeteroPrio variants are able to make a very efficient use of almost all possible configurations of heterogeneous platforms for Cholesky and QR factorizations. Together with the capability of clustering CPU cores, the heuristics that we propose allow to significantly improve the performance of task based applications.

In Chapter 5, we provided theoretical worst-case approximation proofs of HeteroPrio in several cases, including the most general, and we prove that our bounds are tight. Additionally, we have shown experimentally that HeteroPrio produces very efficient schedules for different task graphs, whose makespans are better than the state-of-the-art algorithms from the literature, and very close to the theoretical lower bounds.

Future Work

Our work opens a bridge to close interaction between applications and tasks schedulers. We have shown that providing application specific hints to dynamic schedulers and dynamic corrections to the static schedulers can noticeably improve the 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 run on CPUs", or "these TRSM tasks should run on CPUs", or these tasks should be considered for dynamic corrections etc.

In this thesis we consider static schedules without data transfer costs in different chapters obtained from a constraint program. Formulating data transfer costs adds a lot of constraints to the linear program and CP Optimizer is unable to provide good solutions in limited time. In longer term, we are interested to obtain good static schedules which also take communication costs in to account. It will help us to analyze the behavior of different schedulers in details.

Presently, in HeteroPrio strategy, we do not take communication costs into account while making scheduling decisions. We have evaluated this strategy on a set of dense linear algebra applications, which are compute intensive applications. We want to evaluate the performance of HeteroPrio in less compute applications, where we may have to model the communication costs as well. It would be interesting to study how to combine two completely independent dimensions, *i.e.* , affinity and communication costs in HeteroPrio strategy.

HeteroPrio relies on spoliation mechanism, which requires to abort a running task and restart it on another worker, which is not straightforward to implement and not supported in most state-of-the-art runtime systems. It is possible that aborting a task may take more time than the execution time of that task on current worker. We plan to implement limited lookahead mechanism which removes the need of spoliation from HeteroPrio. A practical implementation of HeteroPrio with lookahead mechanism is currently under way in the StarPU runtime system. We are also interested in an implementation of HeteroPrio where we can use Simgrid with StarPU. The idea is to use precise simulation capability of Simgrid to perform lookahead simulation while making scheduling decisions for ready tasks in StarPU.

In Chapter 4, we use exhaustive search to find the optimal configuration of CPU clusters. How to select the optimal configuration of CPU clusters, when the platform is too large for exhaustive search, would be an interesting area to explore. It would also be interesting to study whether the performance can be improved by changing the clustering of CPUs during the execution instead of using the same configuration from the beginning to the end.

In this thesis, we provide approximation bound of HeteroPrio for a set of independent tasks on two types of resources. We are also interested in the approximation bound of HeteroPrio for general task graphs.

This page is intentionally left blank.

Bibliography

[1] 1979. LINPACK: LINear algebra PACKage.
URL http://www.netlib.org/linpack

[2] 2008. Roadrunner Supercomputer.
URL https://www.top500.org/system/176026

[3] 2009. Avatar.
URL http://www.imdb.com/title/tt0499549/

[4] 2009. ViTE : Visual Trace Explorer.
URL http://vite.gforge.inria.fr

[5] 2011. MORSE: Matrices Over Runtime Systems @ Exascale.
URL https://www.inria.fr/en/associate-team/morse

[6] 2011. MORSE: Matrices Over Runtime Systems @ Exascale.
URL http://icl.cs.utk.edu/morse

[7] 2012. INTRODUCING TITAN: advancing the era of accelerated computing.
URL https://www.olcf.ornl.gov/titan

[8] 2014. Chameleon, A dense linear algebra software for heterogeneous architectures.
URL https://project.inria.fr/chameleon

[9] 2016. Projected Performance Development.
URL https://www.top500.org/statistics/perfdevel

[10] 2016. Sunway TaihuLight.
URL https://www.top500.org/system/178764

[11] 2016. TOP 500 List.
URL https://www.top500.org

[12] Agullo, Emmanuel, Augonnet, Cédric, Dongarra, Jack, Faverge, Mathieu, Ltaief, Hatem, Thibault, Samuel et Tomov, Stanimire,

2011. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. Dans *25th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2011)*. Anchorage, Alaska, USA. doi:10.1109/IPDPS.2011.90.
URL http://hal.inria.fr/inria-00547614

[13] Agullo, Emmanuel, Augonnet, Cédric, Dongarra, Jack, Ltaief, Hatem, Namyst, Raymond, Thibault, Samuel et Tomov, Stanimire, 2010. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. Dans Wen mei W. Hwu, rédacteur, *GPU Computing Gems*, tome 2. Morgan Kaufmann.
URL http://hal.inria.fr/inria-00547847/en/

[14] Agullo, Emmanuel, Aumage, Olivier, Bramas, Berenger, Coulaud, Olivier et Pitoiset, Samuel, 2016. Bridging the gap between OpenMP 4.0 and native runtime systems for the fast multipole method. Research Report RR-8953, Inria.
URL https://hal.inria.fr/hal-01372022

[15] Agullo, Emmanuel, Beaumont, Olivier, Eyraud-Dubois, Lionel et Kumar, Suraj, 2016. Are Static Schedules so Bad? A Case Study on Cholesky Factorization. Dans *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1021–1030. doi:10.1109/IPDPS.2016.90.
URL http://dx.doi.org/10.1109/IPDPS.2016.90

[16] Agullo, Emmanuel, Bramas, Berenger, Coulaud, Olivier, Darve, Eric, Messner, Matthias et Takahashi, Toru, 2016. Task-based FMM for heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, 28(9). doi:10.1002/cpe.3723.
URL https://hal.inria.fr/hal-01359458

[17] Agullo, Emmanuel, Demmel, Jim, Dongarra, Jack, Hadri, Bilel, Kurzak, Jakub, Langou, Julien, Ltaief, Hatem, Luszczek, Piotr et Tomov, Stanimire, 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037+. doi:10.1088/1742-6596/180/1/012037.
URL http://dx.doi.org/10.1088/1742-6596/180/1/012037

[18] Alpatov, Philip, Baker, Greg, Edwards, Carter, Gunnels, John, Morrow, Greg, Overfelt, James, van de Geijn, Robert et Wu, Yuan-Jye J., 1997. PLAPACK: Parallel Linear Algebra Package Design Overview. Dans *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, pages 1–16. ACM, New York, NY, USA. ISBN

0-89791-985-8. doi:10.1145/509593.509622.
URL http://doi.acm.org/10.1145/509593.509622

[19] Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C. et Sorensen, D., 1990. LAPACK: A Portable Linear Algebra Library for High-performance Computers. Dans *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11. IEEE Computer Society Press, Los Alamitos, CA, USA. ISBN 0-89791-412-0.
URL http://dl.acm.org/citation.cfm?id=110382.110385

[20] Augonnet, Cédric, 2011. *Scheduling Tasks over Multicore machines enhanced with accelerators: a Runtime System's Perspective*. Theses, Université Bordeaux 1.
URL https://tel.archives-ouvertes.fr/tel-00777154

[21] Augonnet, Cédric, Thibault, Samuel et Namyst, Raymond, 2009. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. Dans *International Euro-Par Workshops 2009, HPPC'09*, tome 6043 de *Lecture Notes in Computer Science*, pages 56–65. Springer, Delft, The Netherlands. doi:10.1007/978-3-642-14122-5_9.
URL http://dx.doi.org/10.1007/978-3-642-14122-5_9

[22] Augonnet, Cédric, Thibault, Samuel, Namyst, Raymond et Wacrenier, Pierre-André, 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198. doi:10.1002/cpe.1631.
URL http://hal.inria.fr/inria-00550877

[23] Bakhoda, Ali, Yuan, George L., Fung, Wilson W. L., Wong, Henry et Aamodt, Tor M., 2009. Analyzing CUDA workloads using a detailed GPU simulator. Dans *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pages 163–174. doi:10.1109/ISPASS.2009.4919648.
URL http://dx.doi.org/10.1109/ISPASS.2009.4919648

[24] Baptiste, Philippe, Le Pape, Claude et Nuijten, Wim, 2012. *Constraint-based scheduling: applying constraint programming to scheduling problems*, tome 39. Springer Science & Business Media.
URL https://hal.inria.fr/inria-00123562

[25] Bauer, Michael, Treichler, Sean, Slaughter, Elliott et Aiken, Alex, 2012. Legion: Expressing locality and independence with logical

regions. Dans *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11. IEEE Computer Society Press, Los Alamitos, CA, USA. ISBN 978-1-4673-0804-5.
URL http://dl.acm.org/citation.cfm?id=2388996.2389086

[26] Beaumont, Olivier, Cojean, Terry, Eyraud-Dubois, Lionel, Guermouche, Abdou et Kumar, Suraj, 2016. Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources. Dans *International Conference on High Performance Computing, Data, and Analytics (HiPC 2016)*. Hyderabad, India.
URL https://hal.inria.fr/hal-01361992

[27] Beaumont, Olivier, Eyraud-Dubois, Lionel et Kumar, Suraj, 2016. Approximation Proofs of a Fast and Efficient List Scheduling Algorithm for Task-Based Runtime Systems on Multicores and GPUs. Working paper or preprint.
URL https://hal.inria.fr/hal-01386174

[28] Blackford, L. S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. et Whaley, R. C., 1997. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0-89871-397-8.

[29] Bleuse, Raphaël, Gautier, Thierry, Lima, João V. F., Mounié, Grégory et Trystram, Denis, 2014. *Scheduling Data Flow Program in XKaapi: A New Affinity Based Algorithm for Heterogeneous Architectures*, pages 560–571. Springer International Publishing, Cham. ISBN 978-3-319-09873-9. doi:10.1007/978-3-319-09873-9_47.
URL http://dx.doi.org/10.1007/978-3-319-09873-9_47

[30] Bleuse, Raphaël, Hunold, Sascha, Kedad-Sidhoum, Safia, Monna, Florence, Mounié, Grégory et Trystram, Denis, 2016. Scheduling Independent Moldable Tasks on Multi-Cores with GPUs. Research Report RR-8850, Inria Grenoble Rhône-Alpes, Université de Grenoble.
URL https://hal.archives-ouvertes.fr/hal-01263100

[31] Bleuse, Raphael, Kedad-Sidhoum, Safia, Monna, Florence, Mounié, Grégory et Trystram, Denis, 2015. Scheduling Independent Tasks on Multi-cores with GPU Accelerators. *Concurr. Comput. : Pract. Exper.*, 27(6):1625–1638. doi:10.1002/cpe.3359.
URL http://dx.doi.org/10.1002/cpe.3359

[32] Blumofe, Robert D., Joerg, Christopher F., Kuszmaul, Bradley C., Leiserson, Charles E., Randall, Keith H. et Zhou, Yuli, 1995. Cilk:

An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216. doi:10.1145/209937.209958.
URL http://doi.acm.org/10.1145/209937.209958

[33] Blumofe, Robert D et Leiserson, Charles E, 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748.

[34] Bonifaci, Vincenzo et Wiese, Andreas, 2012. Scheduling unrelated machines of few different types. *CoRR*, abs/1205.0974.
URL http://arxiv.org/abs/1205.0974

[35] Bosilca, George, Bouteiller, Aurelien, Danalis, Anthony, Faverge, Mathieu, Haidar, Azzam, Herault, Thomas, Kurzak, Jakub, Langou, Julien, Lemariner, Pierre, Ltaeif, Hatem, Luszczek, Piotr, YarKhan, Asim et Dongarra, Jack, 2011. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. pages 1432–1441. IEEE, Anchorage, Alaska, USA.

[36] Bosilca, George, Bouteiller, Aurélien, Danalis, Anthony, Faverge, Mathieu, Hérault, Thomas et Dongarra, Jack, 2013. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering*. doi:10.1109/MCSE.2013.98.

[37] Bosilca, George, Bouteiller, Aurelien, Danalis, Anthony, Herault, Thomas, Luszczek, Piotr et Dongarra, Jack, 2013. Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach. *Scalable Computing and Communications: Theory and Practice*.

[38] Bouwmeester, Henricus et Langou, Julien, 2010. A critical path approach to analyzing parallelism of algorithmic variants. application to cholesky inversion. *CoRR*, abs/1010.2000.
URL http://arxiv.org/abs/1010.2000

[39] Bouwmeester, Henricus M, 2012. *Tiled algorithms for matrix computations on multicore architectures*. Thèse de doctorat, University of Colorado, Denver.

[40] Broquedis, François, Clet-Ortega, Jérôme, Moreaud, Stéphanie, Furmento, Nathalie, Goglin, Brice, Mercier, Guillaume, Thibault, Samuel et Namyst, Raymond, 2010. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. Dans

IEEE, rédacteur, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Pisa, Italy. doi:10.1109/PDP.2010.67.
URL https://hal.inria.fr/inria-00429889

[41] Brucker, Peter et Knust, Sigrid, 2009. Complexity results for scheduling problems. Web document, URL: http://www2.informatik.uni-osnabrueck.de/knust/class/.

[42] Buttari, Alfredo, Langou, Julien, Kurzak, Jakub et Dongarra, Jack, 2007. Lapack working note 191: A class of parallel tiled linear algebra algorithms for multicore architectures.

[43] Buttari, Alfredo, Langou, Julien, Kurzak, Jakub et Dongarra, Jack, 2008. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590. doi:10.1002/cpe.1301.
URL http://dx.doi.org/10.1002/cpe.1301

[44] Buttari, Alfredo, Langou, Julien, Kurzak, Jakub et Dongarra, Jack, 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53.

[45] Casanova, Henri, Legrand, Arnaud et Quinson, Martin, 2008. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. Dans *10th IEEE International Conference on Computer Modeling and Simulation (UKSim)*.

[46] Chan, Ernie, Van Zee, Field G., Bientinesi, Paolo, Quintana-Orti, Enrique S., Quintana-Orti, Gregorio et Van de Geijn, Robert, 2008. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. Dans *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, page 123–132.

[47] Chandrasekar, J., Kim, I. S., Bernstein, D. S. et Ridley, A. J., 2008. Cholesky-based reduced-rank square-root kalman filtering. Dans *2008 American Control Conference*, pages 3987–3992. doi:10.1109/ACC.2008.4587116.

[48] Chetto, Houssine, Silly, Maryline et Bouchentouf, T, 1990. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194.

[49] Choi, Jaeyoung, Dongarra, Jack, Ostrouchov, Susan, Petitet, Antoine, Walker, David W. et Whaley, R. Clinton, 1996. Design and implementation of the scalapack lu, qr, and cholesky factorization

routines. *Scientific Programming*, 5(3):173–184.
URL http://content.iospress.com/articles/scientific-programming/spr5-3-01

[50] Cojean, Terry, Guermouche, Abdou, Hugo, Andra, Namyst, Raymond et Wacrenier, Pierre-André, 2015. Exploiting two-level parallelism by aggregating computing resources in task-based applications over accelerator-based machines. Inria technical report, Inria.
URL https://hal.inria.fr/hal-01181135

[51] Collange, S., Daumas, M., Defour, D. et Parello, D., 2010. Barra: A parallel functional simulator for gpgpu. Dans *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 351–360. doi:10.1109/MASCOTS.2010.43.

[52] Cosnard, M., Jeannot, E. et Yang, T., 1999. Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors. Dans *Proceedings of the 1999 International Conference on Parallel Processing*, pages 413–421. doi:10.1109/ICPP.1999.797429.

[53] D'Ambra, Pasqua, Guarracino, Mario Rosario et Talia, Domenico, rédacteurs, 2010. *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, tome 6272 de *Lecture Notes in Computer Science*. Springer. ISBN 978-3-642-15290-0.

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

[55] Don Johnston, 2014. HPC Matters to our Quality of Life and Prosperity.
URL http://www.scientificcomputing.com/article/2014/11/hpc-matters-our-quality-life-and-prosperity

[56] Dongarra, Jack, 1988. The linpack benchmark: An explanation. Dans *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474. Springer-Verlag, London, UK, UK. ISBN 3-540-18991-2.
URL http://dl.acm.org/citation.cfm?id=647970.742568

[57] DURAN, ALEJANDRO, AYGUADÉ, EDUARD, BADIA, ROSA M., LABARTA, JESÚS, MARTINELL, LUIS, MARTORELL, XAVIER et PLANAS, JUDIT, 2011. Ompss: A proposal for programming

heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193. doi:10.1142/S0129626411000151.
URL http://www.worldscientific.com/doi/abs/10.1142/S0129626411000151

[58] Dutot, Pierre-Francois, Mounié, Grégory et Trystram, Denis, 2004. Scheduling Parallel Tasks: Approximation Algorithms. Dans Joseph T. Leung, rédacteur, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 26, pages 26–1 – 26–24. CRC Press.
URL https://hal.archives-ouvertes.fr/hal-00003126

[59] Garbow, Burton S., 1974. EISPACK — A package of matrix eigensystem routines. *Computer Physics Communications*, 7(4):179 – 184. doi:http://dx.doi.org/10.1016/0010-4655(74)90086-1.
URL http://www.sciencedirect.com/science/article/pii/0010465574900861

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

[61] Gehrke, Jan Clemens, Jansen, Klaus, Kraft, Stefan E. J. et Schikowski, Jakob, 2016. *A PTAS for Scheduling Unrelated Machines of Few Different Types*, pages 290–301. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-662-49192-8. doi:10.1007/978-3-662-49192-8_24.
URL http://dx.doi.org/10.1007/978-3-662-49192-8_24

[62] Gillman, A., 2011. *Fast direct solvers for elliptic partial differential equations*. Theses, University of Colorado.
URL https://amath.colorado.edu/faculty/martinss/Pubs/

[63] Graham, Ronald L, 1966. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581.

[64] Graham, Ronald L., 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429.

[65] Gustavson, F.G., 2003. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM Journal of Research and Development*, 47(1):31–55. doi:10.1147/rd.471.0031.

[66] Haugh, M., 2004. The Monte Carlo Framework, Examples from Finance and Generating Correlated Random Variables. *Course Notes*.
URL http://www.columbia.edu/~mh2078/MCS04/MCS_framework_FEegs.pdf

[67] Hermann, Everton, Raffin, Bruno, Faure, François, Gautier, Thierry et Allard, Jérémie, 2010. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. Dans *Euro-Par (2)*, pages 235–246.

[68] Higham, Nicholas J., 2002. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd édition. ISBN 0898715210.

[69] Igual, Francisco D., Chan, Ernie, Quintana-Ortí, Enrique S., Quintana-Ortí, Gregorio, van de Geijn, Robert A. et Zee, Field G. Van, 2012. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *J. Parallel Distrib. Comput.*, 72(9):1134–1143. doi:10.1016/j.jpdc.2011.10.014.

[70] Imreh, Csanád, 2003. Scheduling problems on two sets of identical machines. *Computing*, 70(4):277–294. doi:10.1007/s00607-003-0011-9.
URL http://dx.doi.org/10.1007/s00607-003-0011-9

[71] Jaulmes, Luc, Ayguadé, Eduard, Casas, Marc, Labarta, Jesús, Moretó, Miquel et Valero, Mateo, 2015. Exploiting asynchrony from exact forward recovery for due in iterative solvers. Dans *SC '15*, pages 53:1–53:12. ACM, New York, NY, USA. ISBN 978-1-4503-3723-6.

[72] Koelbel, Charles H., 1994. *The High performance Fortran handbook*. Scientific and engineering computation. Cambridge, Mass. MIT Press. ISBN 0-262-11185-3.

[73] Lenstra, Jan Karel, Shmoys, David B et Tardos, Éva, 1990. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*.

[74] Ltaief, Hatem, Gratadour, Damien, Charara, Ali et Gendron, Eric, 2016. Adaptive optics simulation for the world's largest telescope on multicore architectures with multiple gpus. Dans *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '16, pages 9:1–9:12. ACM, New York, NY, USA. ISBN 978-1-4503-4126-4. doi:10.1145/2929908.2929920.
URL http://doi.acm.org/10.1145/2929908.2929920

[75] Manimaran, G et Murthy, C. Siva Ram, 1998. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *Parallel and Distributed Systems, IEEE Transactions on*, 9(3):312–319.

[76] Meike Chabowski, 2016. How HPC Impacts Our Lives II: HPC (and Linux) in the Movies.
URL https://www.suse.com/communities/blog/author/chabowski/

[77] Michael Feldman, 2016. First US Exascale Supercomputer Now On Track for 2021.
URL https://www.top500.org/news/first-us-exascale-supercomputer-now-on-track-for-2021

[78] Michael Feldman, 2017. China will deploy exascale prototype this year.
URL https://www.top500.org/news/china-will-deploy-exascale-prototype-this-year

[79] Ng, Esmond G. et Raghavan, Padma, 1999. Performance of greedy ordering heuristics for sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20(4):902–914. doi:10.1137/S0895479897319313.
URL http://dx.doi.org/10.1137/S0895479897319313

[80] OpenMP Architecture Review Board, 2008. OpenMP application program interface version 3.0.
URL http://www.openmp.org

[81] OpenMP Architecture Review Board, 2013. OpenMP application program interface version 4.0.
URL http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[82] Planas, Judit, Badia, Rosa M, Ayguadé, Eduard et Labarta, Jesus, 2009. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299.

[83] Quach, Willy et Langou, Julien, 2015. A makespan lower bound for the scheduling of the tiled cholesky factorization based on ALAP scheduling. *CoRR*, abs/1510.05107.
URL http://arxiv.org/abs/1510.05107

[84] Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S. et van de Geijn, R. A., 2009. Solving dense linear systems on platforms with multiple hardware accelerators. Dans *PPOPP'09*, pages 121–130.

[85] Quintana-Ortí, Gregorio, Quintana-Ortí, Enrique S, Geijn, Robert A, Zee, Field G Van et Chan, Ernie, 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software (TOMS)*, 36(3):14.

[86] Raghavan, Padma, 1992. *Distributed Sparse Matrix Factorization: QR and Cholesky Decompositions*. Thèse de doctorat, University Park, PA, USA. UMI Order No. GAX92-14255.

[87] Raghavan, Padma, Teranishi, Keita et Ng, Esmond G., 2003. A latency tolerant hybrid sparse solver using incomplete cholesky factorization. *Numerical Linear Algebra with Applications*, 10(5-6):541–560. doi:10.1002/nla.327.
URL http://dx.doi.org/10.1002/nla.327

[88] Rico, Alejandro, Cabarcas, Felipe, Villavieja, Carlos, Pavlovic, Milan, Vega, Augusto, Etsion, Yoav, Ramirez, Alex et Valero, Mateo, 2012. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20. doi:10.1145/2086696.2086715.
URL http://doi.acm.org/10.1145/2086696.2086715

[89] Roberto Viola , 2015. Why do supercomputers matter for your everyday life?
URL https://ec.europa.eu/digital-single-market/en/blog/why-do-supercomputers-matter-your-everyday-life

[90] Rodrigues, A. F., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E. et Jacob, B., 2011. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42. doi:10.1145/1964218.1964225.
URL http://doi.acm.org/10.1145/1964218.1964225

[91] Rothberg, Edward et Gupta, Anoop, 1994. An efficient block-oriented approach to parallel sparse cholesky factorization. *SIAM Journal on Scientific Computing*, 15(6):1413–1439. doi:10.1137/0915085.
URL http://dx.doi.org/10.1137/0915085

[92] Rotkin, Vladimir et Toledo, Sivan, 2004. The design and implementation of a new out-of-core sparse cholesky factorization method. *ACM Trans. Math. Softw.*, 30(1):19–46. doi:10.1145/974781.974783.
URL http://doi.acm.org/10.1145/974781.974783

[93] Sarkar, Vivek, 1989. *Partitioning and scheduling parallel programs for multiprocessing.* Research monographs in parallel and distributed computing. Pitman, London. ISBN 0-273-08802-5.

[94] Shahul, Ahmed Zaki Semar et Sinnen, Oliver, 2010. Scheduling task graphs optimally with a*. *The Journal of Supercomputing*, 51(3):310–332.

[95] Shchepin, Evgeny V. et Vakhania, Nodari, 2005. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters*. doi:http://dx.doi.org/10.1016/j.orl.2004.05.004.

[96] Stanisic, Luka, Thibault, Samuel, Legrand, Arnaud, Videau, Brice et Méhaut, Jean-François, 2014. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. Dans *Euro-par - 20th International Conference on Parallel Processing*.

[97] Susan Blackford, 1997. The Two-dimensional Block-Cyclic Distribution.
URL http://www.netlib.org/scalapack/slug/node75.html

[98] Tomov, Stanimire, Dongarra, Jack et Baboulin, Marc, 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240. doi:10.1016/j.parco.2009.12.005.

[99] Topcuouglu, Haluk, Hariri, Salim et Wu, Min-you, 2002. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274. doi:10.1109/71.993206.
URL http://dx.doi.org/10.1109/71.993206

[100] Ubal, Rafael, Jang, Byunghyun, Mistry, Perhaad, Schaa, Dana et Kaeli, David, 2012. Multi2sim: A simulation framework for cpu-gpu computing. Dans *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 335–344. ACM, New York, NY, USA. ISBN 978-1-4503-1182-3. doi:10.1145/2370816.2370865.
URL http://doi.acm.org/10.1145/2370816.2370865

[101] Vida Glanville, 2015. HPC Short Courses for the UK.
URL http://www2.warwick.ac.uk/fac/cross_fac/hpc-sc/importance

[102] Wu, W., Bouteiller, A., Bosilca, G., Faverge, M. et Dongarra, J., 2015. Hierarchical DAG scheduling for Hybrid Distributed Systems. Dans *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Hyderabad, India.

[103] Xhafa, F., Barolli, L. et Durresi, A., 2007. Immediate mode scheduling of independent jobs in computational grids. Dans *21st International Conference on Advanced Information Networking and Applications (AINA '07)*, pages 970–977. doi:10.1109/AINA.2007.78.

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