HAL
open science

Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures

Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, Jean-François Méhaut

To cite this version:

Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, Jean-François Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. Concurrency and Computation: Practice and Experience, 2015, pp.16. 10.1002/cpe.3555 . hal-01147997

HAL Id: hal-01147997

https://inria.hal.science/hal-01147997

Submitted on 20 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures

Luka Stanisic Samuel Thibault Arnaud Legrand Brice Videau
Jean-François Méhaut

May 7, 2015

Abstract

Multi-core architectures comprising several GPUs have become mainstream in the field of High-Performance Computing. However, obtaining the maximum performance of such heterogeneous machines is challenging as it requires to carefully offload computations and manage data movements between the different processing units. The most promising and successful approaches so far build on task-based runtimes that abstract the machine and rely on opportunistic scheduling algorithms. As a consequence, the problem gets shifted to choosing the task granularity, task graph structure, and optimizing the scheduling strategies. Trying different combinations of these different alternatives is also itself a challenge. Indeed, getting accurate measurements requires reserving the target system for the whole duration of experiments. Furthermore, observations are limited to the few available systems at hand and may be difficult to generalize. In this article, we show how we crafted a coarse-grain hybrid simulation/emulation of StarPU, a dynamic runtime for hybrid architectures, over SimGrid, a versatile simulator of distributed systems. This approach allows to obtain performance predictions of classical dense linear algebra kernels accurate within a few percents and in a matter of seconds, which allows both runtime and application designers to quickly decide which optimization to enable or whether it is worth investing in higher-end GPUs or not. Additionally, it allows to conduct robust and extensive scheduling studies in a controlled environment whose characteristics are very close to real platforms while having reproducible behavior.

1 Introduction

High-Performance Computing architectures now widely include both multi-core CPUs and GPUs. Exploiting the tremendous computation power offered by such systems is however a real challenge. Programming them efficiently is a first concern, but managing the combination of computation execution and data transfers can also become extremely complex, particularly when dealing with multiple GPUs. In the past few years, it has become very common to deal with that through the use of an additional software layer, a runtime system, based on the task programming paradigm [1, 2, 3]. Applications are expressed as a task graph with data dependencies, i.e., a Directed Acyclic Graph (DAG), and provide both CPU and GPU implementations for the tasks. The runtime can then schedule the tasks over all available computation units, and automatically initiate the entailed data transfers. Scheduling heuristics such as HEFT or work stealing are used to automatically optimize that execution [1]. Such runtimes can also take into account the NUMA (Non Uniform Memory Access) effects on architectures with large number of CPUs using shared memory [4, 5]. Application programmers are thus relieved from scheduling concerns and technical details.

As a result, the concern becomes choosing the right task granularity, task graph structure, and scheduling strategies optimizations. Task granularity is of a particular concern on hybrid platforms, since a trade-off must be found between large tasks which are efficient on GPUs but expose little task parallelism, and a lot of small tasks for CPUs. The task graph structure itself can have an influence on execution time, by requiring more or less communication compared to computation, which can be an issue depending on the available bandwidth of the target system. Last but not least, optimizing scheduling strategies has been a concern for decades, and the introduction of hybrid architectures only makes it even more challenging.

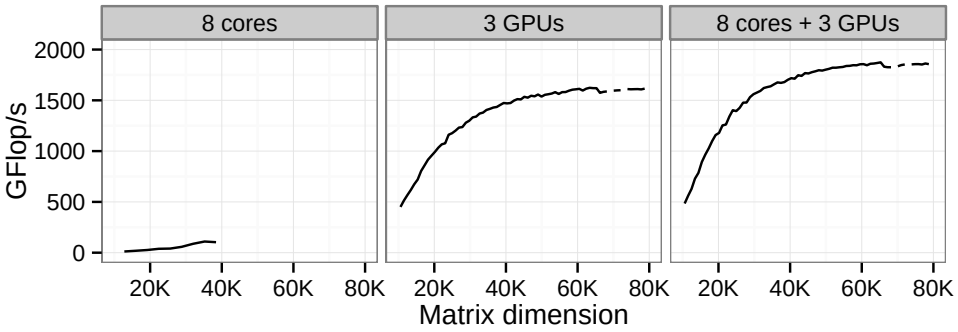


Figure 1: For dense linear algebra applications, most of the processing power is provided by GPUs. These plots depict the performance of the Cholesky application on the Mirage machine (see Table 1). A clearer view of these performance when restricting to CPU resources (8 cores) is provided in Figure 11 (4+4 cores).

Getting accurate measurement results for all combinations is nontrivial and requires reserving the target system for a long period, which can become prohibitive. Moreover, experimenting over a wide range of different platforms is also necessary to make sure that the resulting design choices are generic, and not only suited to the few target systems which were available to developers. Finally, since computation kernel execution time exhibits variability, dynamic schedulers take non-deterministic and opportunistic scheduling decisions. The resulting performance is thus far from deterministic, which makes performance comparisons sometimes questionable and debugging of non-deterministic deadlocks inside such runtimes very hard.

Simulation is a technique that has proven extremely useful to study complex systems and which could be a very powerful way to address these issues. Performance models can be collected for a wide range of target architectures, and then used for simulating different executions, running on a single commodity platform. Since the execution can be made deterministic, experiments become *completely reproducible*, also making debugging a lot easier. Additionally, it is possible to try to extrapolate target architectures, for instance by trying to increase the available PCI bandwidth, the number of GPU devices, etc. and thus even estimate performance which would be obtained on hypothetical platforms. Cycle-accurate simulation of GPUs has hence received a lot of attention recently. However, the current solutions are extremely costly and not precise enough for helping runtime and application designers (see Section 2). Instead, we claim that a top-down modeling approach should be used.

In this article, we show how we crafted a coarse-grain hybrid simulation/emulation of StarPU [1] (see Section 3), a dynamic runtime system for heterogeneous multi-core architectures, on top of SimGrid [6], a simulation toolkit specifically designed for distributed system simulation. Although our work is based on the StarPU runtime system, it could be applied to other runtimes.

For dense linear algebra applications that are studied in this paper, the computational power of GPUs is much higher than the one of CPUs. Figure 1 illustrates such statement by providing the GFlop/s rate for the Cholesky application with StarPU when varying the size of the matrix from 1MB to several GB. All three experimental campaigns were performed on the same machine but used different computation resource sets: 8 cores on the left plot, 3 GPUs on the middle plot, and both CPUs and GPUs on the one on the right. As expected, the hybrid solution provides the best performance and manages to take advantage of both resource types. Nevertheless, since GPUs provide the major fraction of the processing power, we mostly concentrate on executions that use only GPUs for processing. The second part of this article is devoted to explaining how our approach can be extended to fully hybrid setups.

Our contribution are the following:

- we present in detail models that are essential to obtain good performance prediction and quantify their

impact on overall prediction (Sections 5, 6, and 7);

- we validate our models by systematically comparing traces acquired in simulation with those from native executions in a wide variety of settings;
- we show that our approach allows to obtain predictions accurate within a few percents for both Cholesky and LU factorizations on six different NVIDIA GPUs, within a few seconds on a commodity laptop (Section 8 and 8.2);
- we illustrate how it allows to conduct easily preliminary exploratory studies (Section 8.4);
- we demonstrate current limitations of our approach regarding large NUMA machines(Section 8.3);

This article extends the work we previously published in the Euro-Par 2014 conference [7] and which focused on the modeling of GPUs. In this article, we additionally present new results for two additional machines with GPUs, new results on the evaluation of hybrid setups relying on both CPUs and GPUs, and a presentation of the current limitations of our approach to handle NUMA machines comprising a large number of cores.

2 Related Work

In most scientific fields, simulation is used to evaluate complex phenomena and to address all the difficulties raised by the conduction of real experiments such as cost, reproducibility, and extrapolation capability. As a result, many detailed micro-architecture level simulators of GPUs have been developed in the last years. For example GPGPU-Sim [8], one of the most commonly used cycle-accurate GPU simulator, runs directly NVIDIA's parallel thread execution (PTX) virtual instruction set and simulates every detail of the GPU. It is thus very useful for obtaining insights into architectural design problems for GPUs. However, no comparison to an actual GPU is provided in [8] and although the trends predicted by GPGPU-Sim are certainly interesting, it is not clear that it can be used to perform accurate performance prediction of a real hardware. A few other GPU-specific simulators have therefore been developed (e.g., Barra [9] for the NVIDIA G80 or Multi2Sim [10] for the AMD Evergreen GPU). Such specialization allows Multi2sim to report predictions within 5 to 30% of native execution for several OpenCL benchmarks. While this prediction is quite impressive, it comes at the price of a very long simulation time as every detail of the GPU is simulated. The average slowdown of simulations versus native execution is reported to be $44,000\times$ while the one of GPGPU-Sim on a similar scenario is about $90,000\times$[10].

In the context of tuning HPC runtimes, expectations in term of simulation accuracy are extremely high. It is thus difficult to rely on a simulator that may provide the right trends but with a 50% over/under estimation. Choosing the right level of granularity or the correct scheduling heuristic can not be done without precise and quantitative predictions. Such inaccuracies come from an inadequate level of details and can be avoided. Integration of coarse-grain and fine-grain simulations are currently investigated for example within SST [11]. A similar approach is used with TaskSim [12] that has recently been coupled with the NANOS++ runtime system (, on which OmpSs [2] is based) to provide predictions built from multiple levels of abstraction. However, to the best of our knowledge, these approaches address so far only multi-core machines, without GPUs.

Instead, we propose to use a top-down modeling approach such as promoted by the SimGrid project [6], which provides a *versatile* simulation toolkit to study the behavior of large-scale distributed systems like grids, clouds, or peer-to-peer systems. SimGrid builds on fluid network models that have been proven to be a reasonable alternative to both simple analytic models and expensive, difficult-to-instantiate packet-level simulations [13] and has recently been extended to simulate accurately MPI applications on Ethernet networks [14]. In a fluid model, communications, represented by *flows*, are simulated as single entities rather than as sets of individual packets and the bandwidth allocated to flows is constrained by the network resource capacity. While such models ignore all transient phases between two steady-state operation points, they are very flexible and allow to easily account for network topology and heterogeneity as well as many non-trivial

phenomena (e.g., RTT-unfairness of TCP or cross-traffic interferences) [13] at a very low simulation cost. In the next sections, we explain how StarPU has been ported on top of SimGrid and how heterogeneous architectures have been modeled within SimGrid.

3 Porting StarPU over SimGrid

StarPU relies on a task-based abstraction with a clear semantic, which eases the modeling. A StarPU execution consists in scheduling a graph of tasks with data dependencies (i.e., a Directed Acyclic Graph) on the different computing resources, while taking care about data localization. Hence, from the modeling perspective, there are three main components to take into account: StarPU scheduling, computation on the different computing resources, and communication between the computing resources.

Since the StarPU scheduling is dynamic and opportunistic, it is not deterministic and replaying an execution trace, as for example done in BigSim [15] or Dimemas [16] for MPI applications whose control-flow is mostly deterministic, is thus not an option. The most natural approach is thus to execute the StarPU code related to scheduling decisions and to replace actual task execution with SimGrid calls. Yet, to make sure that simulation is carried out in a reproducible and controlled way, SimGrid exports a specific thread API (similar to the POSIX one) that allows the SimGrid kernel to control the scheduling of all application threads. In simulation, such threads run in mutual exclusion and are scheduled upon completion of simulated data transfers and simulated computations. Therefore, any direct regular call to the POSIX threads has to be abstracted as well (e.g., calling `xbt_mutex_acquire()` instead of `pthread_mutex_lock()`). Likewise, in simulation mode, any memory allocation on CPUs or GPUs has to be faked as no actual data processing is done and no actual GPU is necessarily available on simulation machines. They have thus to be replaced by a call to `MSG_process_sleep()` to simulate their overhead. Last, since schedulers may use runtime statistics to take scheduling decisions, time has to be abstracted as well to make sure that simulation time (as returned by `MSG_get_clock()`) is used instead of the system time (as returned by `gettimeofday()`). When running on top of SimGrid, StarPU applications and runtime are thus *emulated* since the actual code is executed, but any operation related to thread synchronization, actual computations of CPU-intensive kernels, or data transfer is in fact *simulated*. More precisely, the control part of StarPU is executed to dynamically inject computation and communication tasks in the simulator. That is the key point that previous emulation attempts from the ICL of the University of Tennessee Knoxville were missing, resulting in ample inaccuracies due to improper synchronization between runtime threads and simulated time.

For simplicity reasons, each core and GPU is represented as a SimGrid host with specific characteristics and it comprises one or several threads which manage synchronization and signaling to StarPU, whenever transfer or computation kernels end. The characteristics of the processing units and of the communication interconnect are measured beforehand on the target machine and expressed in term of processing power, bandwidth, and latency. As a result, such approach is very different from the classical ones described in Section 2 where architecture is modeled in detail and coarse-grain performances are derived from fine-grain simulation of the internals.

Since we fully control the thread execution and timings are solely handled by the simulation, we can decide to include the overhead of the runtime (e.g., the time needed to take scheduling decisions, to manage synchronizations or to manage internal queues) in the simulation or not. In the settings we considered, the runtime overhead is known to be negligible and accounting for it would only make the simulation non-deterministic anymore. Therefore, we decided to ignore it and to only account for the parts related to the application execution. As we will see in the rest of the article, such kind of emulation coupled with a simple modeling of computation and communications may be enough for some applications on some platforms but can lead to gross inaccuracies in others. Showing merely a few examples where simulation and native execution match would hence not be a validation. Instead, we tried to (in)validate our model by conducting as much experiments as possible in a large variety of settings until we find a situation where our simulation fails producing a good prediction. These critical experiments were generally very instructive as they allowed us to understand how to improve our modeling.

In the rest of the article, we present the different sources of errors we identified and the kind of prediction

Table 1: Machines used for the experiments.

Name	Processor	Number of Cores	Frequency	Memory	GPUs
hannibal	Intel Xeon X5550	2×4	2.67GHz	2×24GB	3×QuadroFX5800
attila	Intel Xeon X5650	2×6	2.67GHz	2×24GB	3×TeslaC2050
mirage	Intel Xeon X5650	2×6	2.67GHz	2×18GB	3×TeslaM2070
conan	Intel Xeon E5-2650	2×8	2.0GHz	2×32GB	3×TeslaM2075
frogkepler	Intel Xeon E5-2670	2×8	2.6GHz	2×16GB	2×K20
pilipili2	Intel Xeon E5-2630	2×6	2.6GHz	2×32GB	2×K40
idchire	Intel Xeon E5-4640	24×8	2.4GHz	24×31GB	/

that can be done once they are fixed.

4 Experimental Setting

We conducted series of experiments to (in)validate our modeling approach. All conclusions were drawn from analyzing and comparing GFlop/s rate, makespans and traces of StarPU on one hand (called *Native* in the following), and StarPU on top of SimGrid (called *SimGrid* in the following) on the other.

Before running applications, StarPU needs to obtain a calibration of the platform, which consists in measuring bandwidths and latencies for communication between each processing unit, together with evaluating timings of computation kernels [17]. Such information is used to guide StarPU schedulers' decisions when delegating tasks to available workers. StarPU has thus been extended to generate at the same time a (XML) SimGrid description of the platform, which can later be used for simulation purposes. It is important to understand that only the calibration, which is meant to be run once and for all on the target system before conducting any performance investigation, is used in the *SimGrid* simulation and that it is not linked to the application being studied. The only condition is of course that the application can use only computation kernels that have been measured. Such a clear separation allowed all the simulations presented in this paper to be performed on personal commodity laptops. This separation also allows to simulate machines we don't have access to, knowing merely their characteristics (i.e., computation kernel runtimes and memory bandwidth).

To study the validity of our models, we used the systems described in Table 1. All six NVIDIA GPUs have distinct characteristics and span three different generations, which intends to demonstrate the validity of our approach on a range of diverse machines. Additionally, we have experimented on a large NUMA machine without GPUs, but with 24 nodes each having 8 cores.

Regarding applications, we decided to focus on two common dense linear algebra kernels: *Cholesky* and *LU* factorization.

Concerning task granularity, for the executions running on GPUs we fixed a relatively large block size (960×960) as it is representative of what is typically used to achieve good performances. In the first series of experiments we present, CPUs were only controlling the execution and scheduling of tasks while GPUs had the roles of workers, meaning that the whole computation was done entirely on multiple GPUs. We focused on this kind of scenario as GPUs have stable performance and provide a significant fraction of computational power in dense linear algebra. However, in Section 8.3 we report experiments conducted on a NUMA machine with a large number of cores with no GPUs and for which we thus used a smaller block size (320×320) that is much better suited to CPU resources. Finally, we also investigated situations involving both CPUs and GPUs at the same time.

This whole work was done in the spirit of open science and reproducible research. Both StarPU and SimGrid software are free software available online. All the source code and experiment results presented in this paper are publicly available[1]. Supplementary data, which is not presented in this paper due to space limitation, are also available at the same location along with all the scripts, raw data files and traces which

[1] http://starpu-simgrid.gforge.inria.fr/

Table 2: Typical duration of runtime operations.

Operation	Transfer queue management	GPU memory allocation (`cudaMalloc`)	GPU memory deallocation (`cudaFree`)	Pinned RAM allocation (`cudaHostAlloc`)
Time	10μs	175μs	125μs	650μs/MB

allow to regenerate this document. The methodology used during the whole project is a good example of conducting an exhaustive, coherent and comprehensible research, and is explained in more detail in [18].

Finally, assessing the impact of our various modeling attempts is quite difficult. Some of them are specifically linked to the modeling of the StarPU runtime, while others are more linked to the modeling of communications or to the computation variability. Obtaining a good predictive power is the combination of a series of improvements. Hence, comparing different runtime modeling options with a native execution while having a poor modeling of communications and computations would not be very meaningful. So instead, we evaluate our different runtime modeling options while using the best options for communication and computation modeling. Likewise, when we evaluate various communication modeling options, we always use the best modeling option of runtime and computations, which allows us to evaluate how much accuracy we may lose by overlooking this particular aspect.

5 Modeling runtime system

Since StarPU is dynamic, inaccurate emulation of the control part would produce different scheduling decisions and would damage prediction of the overall execution time. We show how, in some cases and if not treated correctly, this can produce misleading results, and present how these issues were eliminated.

As we already mentioned, process synchronizations, memory allocations of CPU or GPU, submission of data transfer requests are all faked in simulation mode, whereas such operations in native execution do take time and have an impact on the overall performance. Several delays were included in the simulation to account for their overhead (Table 2 depicts typical duration of such operations). It is interesting to note that pinned RAM allocation is linear with memory size since it has to pin each physical page of the allocation, while other allocations have more standard, constant costs. Another (probably the most) influential parameter for accurate modeling of runtime proved to be the size of GPU memory. Such hardware limits force the scheduler to swap data back and forth between the CPUs, main memory and GPUs. These data movements saturate the PCI bus, producing a tremendous impact on overall performance. It is thus critical to keep track of the amount of memory allocated by StarPU during the simulation to make sure the scheduler will behave in the same way for both real native executions and simulations.

Figure 2 illustrates the importance of taking into account the runtime parameters described above. Each curve depicts GFlop/s rate of experiments for 72 different matrix dimensions (the matrix dimension 80,000 corresponds to ≈25GB). The *Native* solid line shows the execution of StarPU on the native machine, while the other two are the results of the simulation: *naive* for execution without any runtime adjustments and *smart* with all of them included. The left plot depicts a situation where all these optimizations have very little influence as both *naive* and *smart* lines are almost overlapping with the *Native* line. On the other hand, for some other combinations of machines and applications (right plot), having a precise modeling of runtime is critical as otherwise, simulation may highly overestimate the performance for large matrix sizes. Nonetheless, we remind that the excellent predictions achieved in these examples are also the result of the careful modeling of communications and computations, which we will present in the next sections.

6 Modeling communication in hybrid systems

Due to the relatively low bandwidth of the PCI bus, applications running on hybrid platforms often spend a significant fraction of the total time transferring data back and forth between the main RAM and the

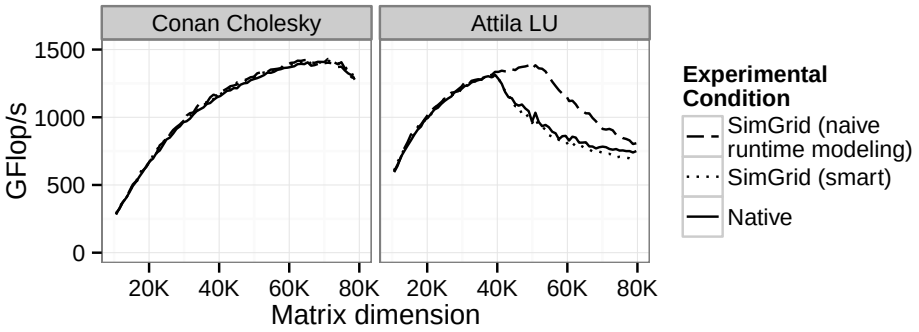
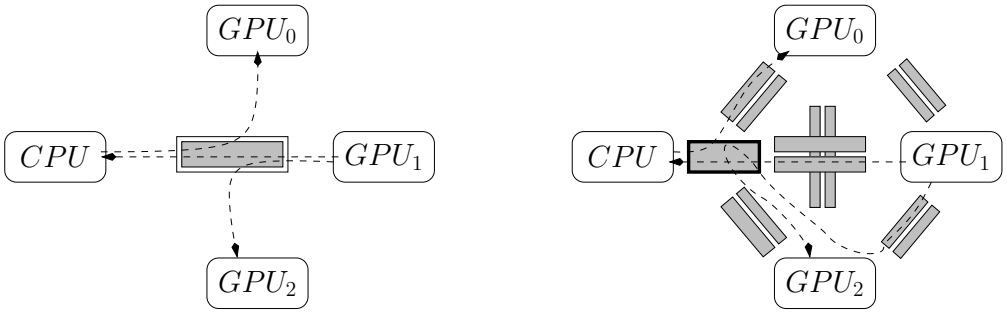


Figure 2: Illustrating the influence of modeling runtime. Careless modeling of runtime may be perfectly harmless in some cases, it turns out to be misleading in others.



(a) Crude modeling

(b) More elaborated modeling

Figure 3: Communication and topology modeling alternatives. In the crude modeling, a single link is used and communications do not interfere with each others. The more elaborated modeling allows to account for both the heterogeneity of communications and the global bandwidth limitation imposed by the PCI bus.

GPUs. Modeling communication between computing resources is thus of primary importance. As a first approximation (see Figure 3(a)), the transfer time between resources could be modeled as a single link with a latency and a transfer rate corresponding to typical characteristics of the PCI bus. However, such modeling does not account for many architectural aspects. First, the bandwidth between CPU and GPU is asymmetrical. Second, communication characteristics are not uniform among all pairs of CPUs and GPUs, as it depends on the chipset architecture. We decided to account for it by using a dedicated uplink and a downlink with different characteristics for each pair of resources (see Figure 3(b)). Furthermore, any communication between two resources has to go through a common shared link (in bold), which represents the maximum capacity of the PCI bus. Modeling contention in such a way greatly improves on the previous approach but ignores that depending on resources involved in a communication, data transfers may be serialized or not. For example, although most CUDA transfers are serialized whenever they involve the same resource, on some systems it is possible to transfer both from GPU_0 to GPU_1 and from GPU_1 to GPU_0 at the same time. Such serializations have thus also to be measured in the calibration phase.

Additionally, to move chunks of matrices between resources, StarPU relies on the `cudaMemcpy2D` function. The performance of this function is not exactly the same as the one of `cudaMemcpy`, which was used in the original calibration process. More important, it turns out that the pitch (i.e., the stride of the original matrices) can have a significant impact on transfer time on some GPUs (see Figure 4) whereas it can be relatively safely ignored on others. Therefore, communication time is modeled as a piece-wise linear function of data payload whose slope and intercept depend on the pitch of the matrix.

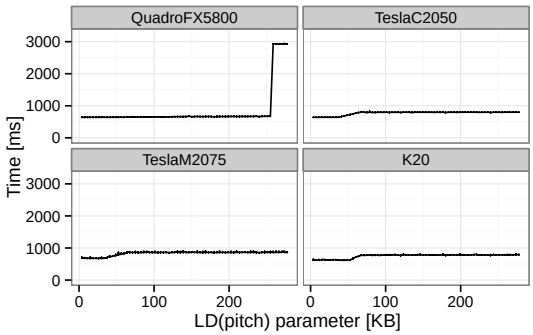


Figure 4: Transfer time of 3,600 KB using `cudaMemcpy2D` depending on the pitch of the matrix.

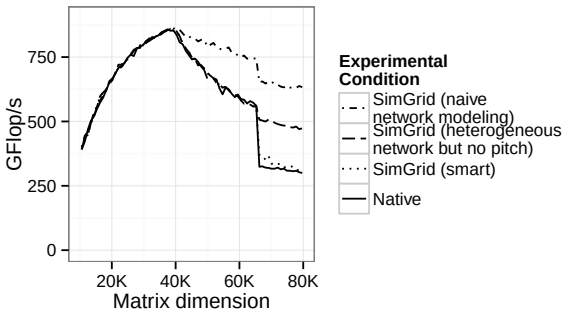


Figure 5: Performance of the LU application on Hannibal (QuadroFX5800 GPUs) using different modeling assumptions.

Again, for a given application and a given target architecture, it may not be necessary to take care of all such details to obtain a good prediction. For example, as illustrated on Figure 5, a naive network modeling such as the one on Figure 3(a) proved excellent predictions when matrix dimension is smaller than 40,000. Beyond such size, a more precise modeling of the network (as in Figure 3(b)) is necessary. Beyond 66,240, the behavior of `cudaMemcpy2D` changes drastically and has to be correctly modeled to obtain a good prediction of the performances.

A possible way to improve further might be to use more elaborate performance models for CPU-GPU data transfers that work with arbitrary data size [19]. However, in the applications we used so far, matrices are split in equal blocks (e.g., 960×960) and every communication is just an array of transfers of such blocks. Since the size of the block does not change during the execution, simulation requires precise models only for that block size. The main difficulty when simulating these applications is thus more related to correctly modeling PCI bus contention that many single transfers will produce and this is one of the strongest points of SimGrid.

7 Modeling computation

When running a simulation, the actual result of the application is of no interest. Hence the execution of each kernel is replaced by a virtual delay accounting for its duration. For dense linear algebra applications such as the ones considered in this article, the parameters of computation kernels is fixed by the task granularity throughout the whole execution of the application. Therefore, kernel duration on a given computation resource is well modeled by a fixed probability distribution. In our initial approach, we used the mean duration of each computation kernel, which was benchmarked by StarPU during the calibration phase. In such context, modeling kernels with a single constant is accurate enough for the intended purposes and the mean value proved to be a good representative in our experiments. However, using a fixed value leads to a deterministic schedule in simulation, which may bias the simulation and does not allow to verify the ability of the scheduling algorithms to handle resource variability.

Therefore, we modified StarPU to capture the timing of every computation during a native execution. Such collection of data can then be used to analyze the computation time distribution which can be approximated using irregular histograms [20]. Regular ones (with uniform bin-width) revealed very inefficient at representing details of distributions where a small but non-negligible fraction of values are an order of magnitude larger than the vast majority of measurements. Such approximation can then be used in the simulation by generating pseudo-random variables from the histograms.

Although this technique allows to obtain different simulated schedules by changing the seed of the simu-

lation, no significant gain in term of accuracy could be observed for the applications and machines we used so far. The makespan is always very similar in both cases (mean duration vs. random duration following an approximation of the original distribution). Nonetheless, we strongly believe that in some more complex use cases, e.g., sparse linear algebra algorithms, using fine models like histograms may provide more precise predictions. Taking into account the dependence on kernel parameters (as done for example in [21]) that will vary throughout the application execution will then also be critical.

8 Prediction Accuracy in a Wide Range of Settings

As we explained in the previous sections, a careless modeling of any aspect of runtime, communications or computations, can lead to gross inaccuracies for particular combinations of machines and applications. We show in this section that we managed to cover the most important issues, which enables us to obtain excellent prediction of performances. First we display results on the machines with the wide variety of GPUs, which are the scenarios we mostly focused on during this research. Second, we show how our methods could be equally applied to the executions that are using both CPUs and GPUs for doing computation. Then, we exhibit current limits of our work regarding large NUMA machines, for which we still do not have sound models. Finally, we demonstrate a typical use case with different scheduling algorithms, where StarPU users already benefit from our solution since simulations can be run on their personal machines instead of having to conduct native experiments on remote clusters, which saves a lot of time and computing resources.

8.1 Accurate Performance Predictions for Hybrid Machines

Figure 6 depicts the performance as a function of the size of the matrix for the two applications LU and Cholesky and for the six different hybrid systems we described in Table 1 (the last Idchire machine does not have GPUs). For all combinations, the prediction obtained with SimGrid is very accurate. There are a couple of scenarios for which the error is larger than a few percents but such discrepancies are actually due to the fact that the prototype experimental machines are sometimes perturbed by other users, OS, etc. Regardless of that, these errors stay always lower than 6%, which is still very precise. Additionally, the trend is perfectly predicted as well as the size beyond which performance drops.

A closer look at traces allows to see that this approach does not only provide a good estimation of the total runtime but also offers an accurate simulation of the scheduling details. In Figure 7, we compare traces from *Native* execution with *SimGrid* simulation, focusing only on the most important states. *DriverCopy* corresponds to the CPU managing a data transfer to the GPU, while *POTRF*, *TRSM* and *GEMM* are the three kernels that compose of the *Cholesky* application. One can observe that GPUs perform all the computations, while CPUs provide them with data. Additionally, CPU0 is responsible for doing all the scheduling. Since even with the same parameters, native traces differ from one execution to another, a point-to-point comparison with a simulation trace would not make sense. However, we can check that both the *Native* and the *SimGrid* traces are extremely close, which allows to study and understand the potential weaknesses of a scheduler.

8.2 Using both CPUs and GPUs for Computation

It was illustrated in the introduction (see Figure 1) that CPUs do not have such a major influence on the performance of the dense linear algebra kernels used in this paper. However, this may be different for other kind of applications with less optimized kernels. Therefore, we also investigated the situation where both CPUs and GPUs are used as computation resources. The resulting prediction are depicted in Figure 8, which shows once again how our *SimGrid* predictions compare favorably with real experiments in hybrid setups.

Again, to illustrate the fact that not only the makespan is accurately predicted but that the whole scheduling is correctly modeled, we compare two execution traces in Figure 9. These traces correspond to an experiment performed on the Mirage machine that has 3 GPUs and 12 cores. 8 of them where used for doing computations while 3 for were dedicated to GPU transfer management and 1 was dedicated to scheduling,

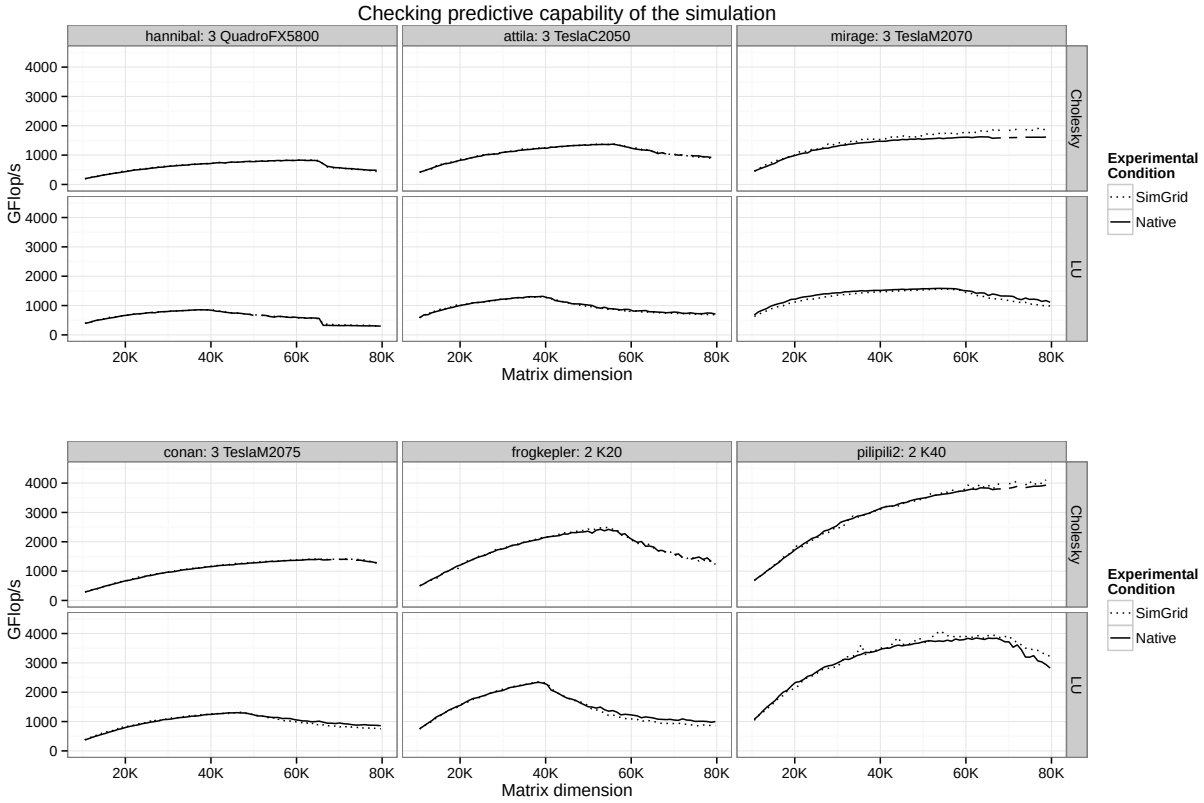


Figure 6: Checking predictive capability of our simulator in a wide range of settings.

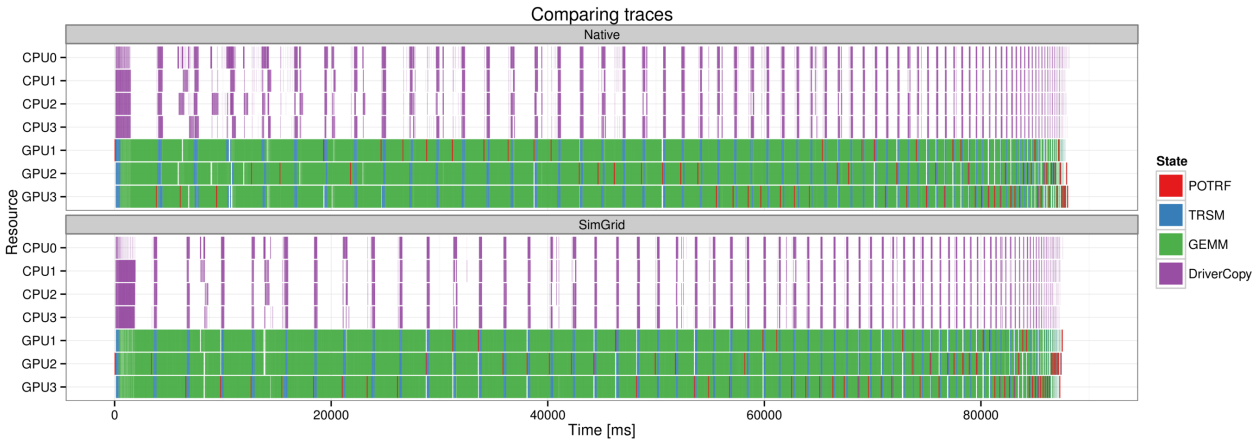


Figure 7: Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $72,000 \times 72,000$ matrix on the Conan machine but using only GPU resources for processing the application.

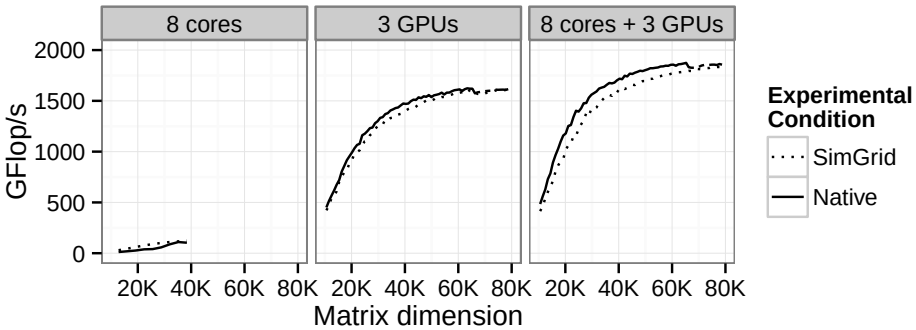


Figure 8: Illustrating simulation accuracy for Cholesky application using different resources of the Mirage machine.

transfers and control. These traces can be compared to the ones of Figure 7, since in both cases, the same application and matrix size are used. Only the GPU slightly differs. The conclusion is that adding 8 CPUs improved the performance by approximately 20% and such conclusion can be drawn solely on simulations.

To convince the readers even further, we provide another trace comparison in Figure 10, again based on the same experimental setup (the Mirage machine using 8 cores for computations, 3 for GPU transfers and 1 for scheduling) and application (Cholesky) only using a different matrix size and a different implementation of the kernels. In this execution we did not use the Intel Math Kernel Library (MKL) but simple non-over optimized ones and thus executing kernels on CPUs was 10 times slower than in all other results from CPUs presented in this article. Although such results are not necessarily interesting in term of performance, we still think that it is important to show that we manage to obtain accurate performance prediction in such context as well since not all users may have access to the proprietary Intel libraries.

The general shape of the schedule of Figure 10 is the same in both *Native* and *SimGrid* traces and one can observe several characteristics of the scheduling algorithm:

- The shortest kernel (*POTRF*) was executed mostly on CPU0, except in the very beginning and at the end where it was executed on GPUs. This is due to the fact that although the execution of POTRF is faster on the GPUs, GPUs are relatively more efficient for GEMM operations than CPU resources. The GPU resources should thus not be "wasted" for such kernels when the application reaches steady-state. However, using the GPUs for such kernels at the beginning and at the end of the schedule makes sense, since it allows to release available tasks as soon as possible in the beginning and to improve the execution of the critical ones in the end.

- The *TRSM* kernels are executed on every CPU worker in the first part of the execution, while they are performed regularly and much faster on the GPUs in the rest of the execution.

All these phenomena are also present in the *SimGrid* trace. As expected, scheduling is not identical, since StarPU is dynamic and with two native executions with the same parameters, traces would not be exactly the same. For example, there is a slight difference in the distribution (the number of times) of TRSM kernels' allocation between CPUs and GPUs. It can be explained by the fact that the execution time variability of this kernel was not accounted for in this simulation. There was thus interest for the scheduler to execute 9 series of such kernels on the CPUs in simulation although only 7 of them were done on the CPUs in the native execution. Note that this number varies from one native run to the other and that the simulation is thus only slightly idealizing the real conditions in a deterministic way. However, all the trends of the real execution are correctly accounted by the simulation.

8.3 NUMA machine

A potential shortcoming of our model could be NUMA architectures. When executing an application using only CPU resources, there is no explicit data transfers, as the workers use shared memory to exchange data. Yet, the time to access the shared data depends on the used memory banks. Additionally, effective memory bandwidth depends on efficient utilization and it is particularly sensitive to suboptimal block size and memory strides. Such aspects are extremely difficult to model and are currently ignored in our simulations. Although this is not too harmful for systems with relatively few cores (like the ones we used in the previous experiments), it can be much more annoying with larger architectures.

In the following, we present a set of experiments that illustrates the impact of the NUMA effect on the realism of our predictions. On Figure 11, we present two different experimentation setups on the same Mirage machine, which has 12 cores that are distributed on two NUMA nodes (6+6). In these experiments, we used only 8 cores, since the other 4 are normally in hybrid executions dedicated to GPU transfers and scheduling as it is generally how the best performances are obtained in practice. The plot on the left use an improper balancing of computing threads as 6 threads are pinned to one node and only 2 on second node. On the other hand, the plot on the right is well balanced (4+4). Since the simulation does not currently take the NUMA topology into account, the *SimGrid* prediction is identical in both the left and the right plot. As one could however expect, resources are more efficiently used in the second case and thus the performance

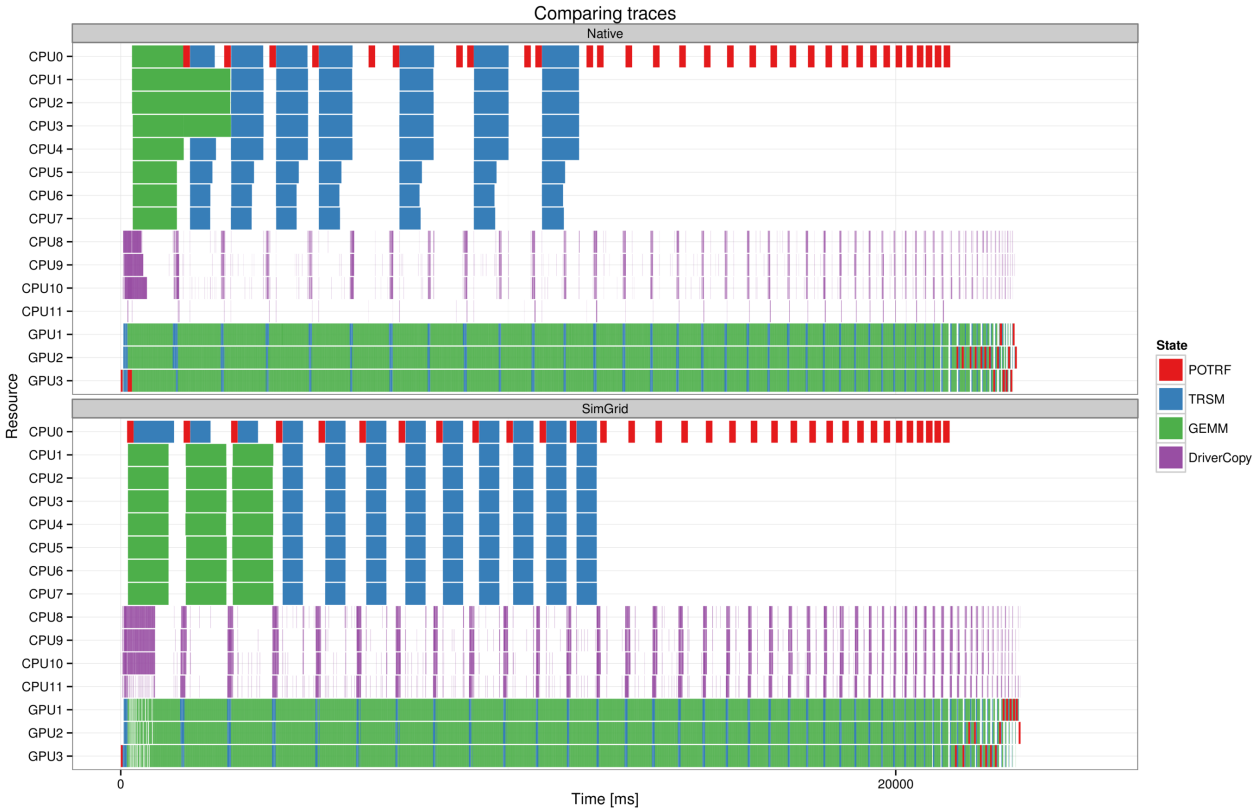


Figure 10: Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $48,000 \times 48,000$ matrix on the Mirage machine using 8 cores and 3 GPUs as workers. Executing kernels on CPUs is much longer since Intel MKL libraries were not used, however simulation predictions are still very precise.

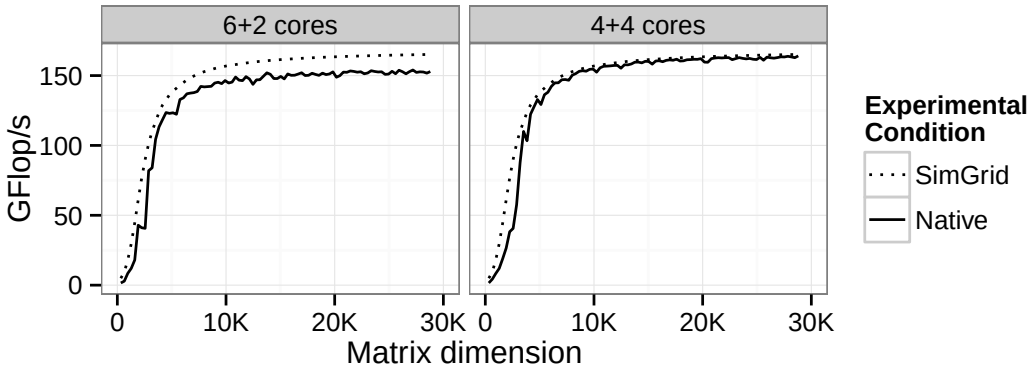


Figure 11: Illustrating the impact of deployment when using 8 cores on two NUMA nodes on the Mirage machine.

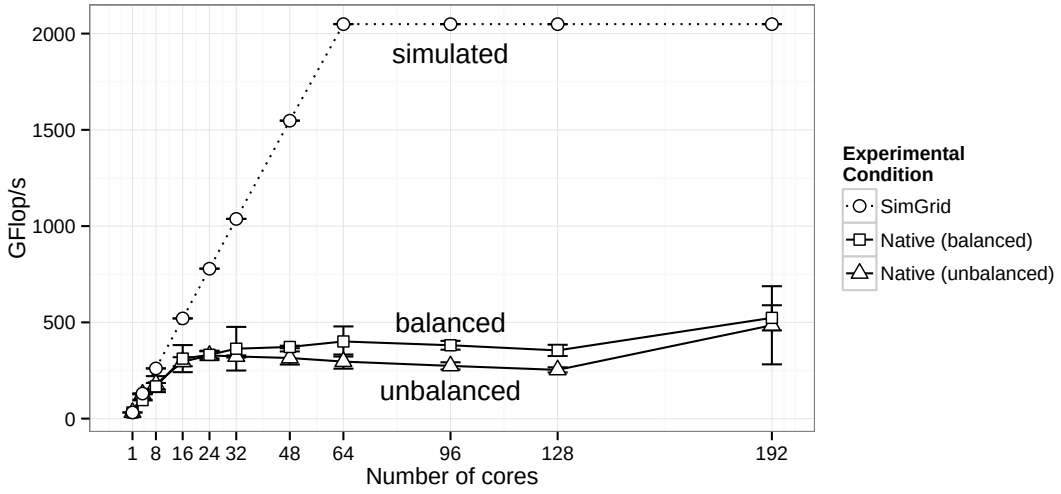


Figure 12: Simulation predictions of Cholesky application with a $32,000 \times 32,000$ matrix (block size 320×320) on large NUMA Idchire machine are precise for a small number of cores, but scale badly. The reason is that the memory is shared, while models are not taking into account various NUMA effects.

of *Native* executions are better in the second scenario and match the *SimGrid* predictions. In such small platforms, our approach can thus provide a sound estimation of the performance that one should expect but cannot account for the performance loss due to a bad deployment. In more extreme setups, our predictions are however likely to be too optimistic.

To illustrate even further such difficulty, we conducted a similar experiment on Idchire that has 24 NUMA nodes each with 8 cores. On Figure 12, one can once again observe that there is a significant difference in terms of performance between a good and a bad balancing of the cores for the native executions. On such platforms, our SimGrid results provide decent predictions only for execution with a very small number of cores, while for the other setups it greatly overestimates the capabilities of the system. This is explained by our current inability to account for the performance degradation of interfering memory-bound kernels, the NUMA effects and inter-node traffic.

These are known limits of our approach that may be overcome by keeping track of data localization and trying to model data movement more precisely. However, although the performance loss incurred by interfering kernels that contend on the memory hierarchy can be measured, it is quite difficult to model. We are thus still investigating how to account for such situation.

8.4 Typical Use Case: Scheduling Studies

One of the main challenges that StarPU developers encounter is how to efficiently exploit all the available heterogeneous resources. To do so, they develop different scheduling techniques, that may be specifically tailored for a given type of machine architectures.

For example, the reason for the performance drop observed on Figure 6 and which is more and more critical with newer GPUs can be explained by the need to move data back and forth between the GPUs and the main memory whenever matrix size exceeds the memory size of the GPUs. The scheduler we used in Figure 6 is the *DMDA* (*Deque Model Data Aware*) scheduler. Although it schedules tasks where their termination time (including data transfer time) will be minimal, it does not take care of the number of available data buffers on each GPU. Such greedy strategy may be harmful as one GPU may be overloaded with work and forced to evict some data, as it cannot handle the whole matrix. Two other strategies *DMDAR*

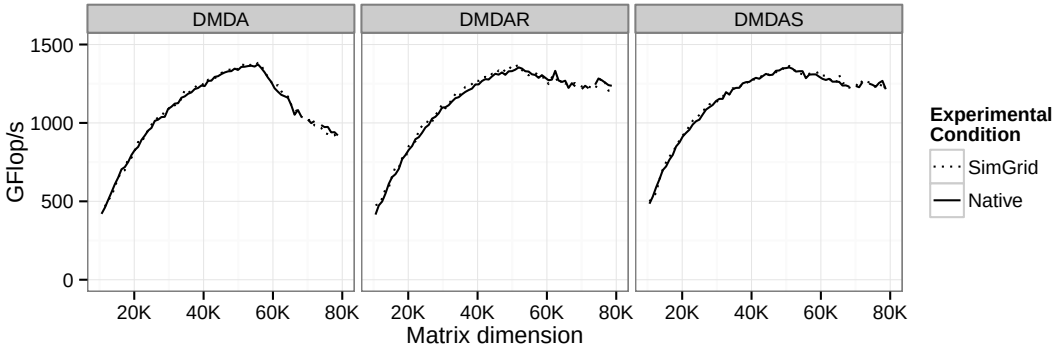


Figure 13: Cholesky on Attila: studying the impact of different schedulers.

and *DMDAS* were designed to execute in priority tasks whose data is already on the GPU, before tasks whose data is not yet available.

Therefore, we decided to check whether these two other schedulers could stabilize performances at the peak or not. To this end, we first ran the corresponding simulations and obtained a positive answer (Figure 13). Later, when the target system became accessible again, we confirmed these results by running the same experiments and as can be seen on Figure 13, our simulations were again perfectly accurate.

Researchers studying different scheduling algorithms on the StarPU implementation of MAGMA/MORSE[2] applications also benefit from this simulation approach. They have started to use extensively SimGrid simulations for screening experiments [22]. Indeed a major advantage of doing simulations rather than running the real experiments, is that simulations are fast, reproducible (, which simplifies the analysis) and do not require an access to the remote experimental cluster. Since our simulations provides reliable predictions, it is possible to screen a wide range of parameters and quickly see whether a given approach seems effective or not. Such screening thus helps refining the set of configurations that are worth being tested in real environments.

8.5 Time to Simulate

It is important to mention that the time to run each simulation typically takes few seconds compared to sometimes several minutes for a real experiment. Compared to architecture-level simulators (see Section 2) whose average slowdown of simulations versus native execution is of the order of magnitude of several dozens of thousands, our coarse-grain simulation allows to obtain a speedup of ten to a hundred depending on the workload and on the speed of the machine. Furthermore, since the target system is not required anymore, it is easy to run series of simulations in parallel.

9 Conclusion and Future work

In this article, we have explained how to model and simulate using SimGrid a task-based runtime system on a hybrid multi-core architecture comprising several GPUs. Unlike fine-grain GPU simulators that have been proposed in the past and which focus on architectural details of GPUs, our coarse-grain approach allows to accurately predict the actual running time and to perform extremely quickly extensive simulation campaigns to study various alternatives. We demonstrated the precision of our simulations using the critical method, i.e., by testing our models and by conducting as much experiments as possible in a large variety of settings (two standard dense linear algebra applications, six different generations of GPUs, several scheduling

[2] http://icl.cs.utk.edu/projectsdev/morse/

algorithms) until we find a situation where our simulation fails at producing a good prediction, in which case we fixed our modeling. Such a tool is extremely interesting for both StarPU developers and users as it allows (i) to easily and accurately evaluate the impact of various parameters or scheduling alternatives (ii) to tune and debug applications on a commodity laptop (instead of requiring a dedicated access to a high-end machine) *in a reproducible way* (iii) to obtain reliable comparison of performance estimations that may allow to detect problems with some real experiments (perturbation, configuration issue, etc.).

Now that we have proven the efficiency of this approach on dense linear algebra kernels, we intend to continue with this work in two directions. First, StarPU was recently extended to exploit clusters of hybrid machines by relying on MPI [23]. Since SimGrid's ability to accurately simulate MPI applications has already been demonstrated [14], combining both works should allow to obtain good performances predictions of complex applications on large-scale high-end HPC infrastructures. Such approach may also provide a solution for the NUMA architectures since it will make data transfers explicit at a negligible cost if well implemented. Second, many numerical applications have been recently ported on top of StarPU, including sparse linear algebra (QR-MUMPS [24]) and FMM methods. Such applications are less regular and are thus likely to be more challenging to model. However, a reliable performance evaluation methodology would bring considerable insights to the developers.

Acknowledgements

This work is partially supported by the SONGS ANR project (11-ANR-INFRA-13). We warmly thank Paul Renaud-Goud for his help with the initial investigation of validity, Emmanuel Agullo for motivating this study and providing insights on its usefulness, and finally Augustin Degomme for helping to deal with numerous technical challenges. Some of the experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS.

REFERENCES

[1] Augonnet C, Thibault S, Namyst R, Wacrenier PA. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* Feb 2011; **23**:187–198.

[2] Ayguadé E, Badia RM, Igual FD, Labarta J, Mayo R, Quintana-Ortí ES. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. *Proceedings of the 15th Euro-Par Conference*, 2009.

[3] Bosilca G, Bouteiller A, Danalis A, Herault T, Lemarinier P, Dongarra J. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. *IEEE International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, 2011; 1151–1158.

[4] Pérache M, Jourdren H, Namyst R. MPC: A unified parallel runtime for clusters of NUMA machines. *Euro-Par 2008 – Parallel Processing*, *Lecture Notes in Computer Science*, vol. 5168, Luque E, Margalef T, Benítez D (eds.). Springer Berlin Heidelberg, 2008; 78–88.

[5] Broquedis F, Furmento N, Goglin B, Namyst R, Wacrenier PA. Dynamic task and data placement over NUMA architectures: An openMP runtime perspective. *Evolving OpenMP in an Age of Extreme Parallelism*, *Lecture Notes in Computer Science*, vol. 5568, Müller M, de Supinski B, Chapman B (eds.). Springer Berlin Heidelberg, 2009; 79–92.

[6] Casanova H, Giersch A, Legrand A, Quinson M, Suter F. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing* Jun 2014; **74**(10):2899–2917.

[7] Stanisic L, Thibault S, Legrand A, Videau B, Méhaut JF. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-core Architectures. *Euro-Par*, 2014; 50–62.

[8] Bakhoda A, Yuan GL, Fung WWL, Wong H, Aamodt TM. Analyzing CUDA workloads using a detailed GPU simulator. *ISPASS*, 2009; 163–174.

[9] Collange S, Daumas M, Defour D, Parello D. Barra: A Parallel Functional Simulator for GPGPU. *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication*, 2010; 351–360.

[10] Ubal R, Jang B, Mistry P, Schaa D, Kaeli D. Multi2Sim: A Simulation Framework for CPU-GPU Computing. *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, ACM: New York, NY, USA, 2012; 335–344.

[11] Rodrigues AF, Hemmert KS, Barrett BW, Kersey C, Oldfield R, Weston M, Risen R, Cook J, Rosenfeld P, CooperBalls E, *et al.*. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.* Mar 2011; **38**(4):37–42.

[12] Rico A, Cabarcas F, Villavieja C, Pavlovic M, Vega A, Etsion Y, Ramírez A, Valero M. On the simulation of large-scale architectures using multiple application abstraction levels. *TACO* 2012; **8**(4):36.

[13] Velho P, Schnorr L, Casanova H, Legrand A. On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations. *ACM Transactions on Modeling and Computer Simulation* Oct 2013; **23**(3).

[14] Bedaride P, Degomme A, Genaud S, Legrand A, Markomanolis G, Quinson M, Stillwell, Lee M, Suter F, Videau B. Toward Better Simulation of MPI Applications on Ethernet/TCP Networks. *4th International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS)*, 2013.

[15] Zheng G, Kakulapati G, Kalé L. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[16] Badia RM, Labarta J, Giménez J, Escalé F. Dimemas: Predicting MPI Applications Behaviour in Grid Environments. *Proc. of the Workshop on Grid Applications and Programming Tools*, 2003.

[17] Augonnet C, Thibault S, Namyst R. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. *3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, 2009.

[18] Stanisic L, Legrand A, Danjean V. An Effective Git And Org-Mode Based Workflow For Reproducible Research. *ACM SIGOPS Operating Systems Review* 2015; **49**:61–70.

[19] van Werkhoven B, Maassen J, Seinstra FJ, Bal HE. Performance models for CPU-GPU data transfers. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, IL, USA, May 26-29, 2014*, 2014; 11–20.

[20] Denby L, Mallows C. Variations on the Histogram. *Journal of Computational and Graphical Statistics* 2009; **18**(1):21–31.

[21] Clarke D, Zhong Z, Rychkov V, Lastovetsky A. Fupermod: a software tool for the optimization of data-parallel applications on heterogeneous platforms. *The Journal of Supercomputing* 2014; **69**(1):61–69.

[22] Agullo E, Beaumont O, Eyraud-Dubois L, Herrmann J, Kumar S, Marchal L, Thibault S. Bridging the gap between performance and bounds of cholesky factorization on heterogeneous platforms. *Proceedings of the 24th International Heterogeneity in Computing Workshop (HCW'15)*, 2015.

[23] Augonnet C, Aumage O, Furmento N, Namyst R, Thibault S. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface (EuroMPI)*, Springer-Verlag, 2012; 298–299.

[24] Buttari A. Fine granularity sparse QR factorization for multicore based systems. *Applied Parallel and Scientific Computing*, *Lecture Notes in Computer Science*, vol. 7134, Jónasson K (ed.). Springer Berlin Heidelberg, 2012; 226–236.