HAL
open science

Modeling and Simulation 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. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. Euro-par - 20th International Conference on Parallel Processing, Aug 2014, Porto, Portugal. pp.50-62, 10.1007/978-3-319-09873-9_5 . hal-01011633

HAL Id: hal-01011633

https://inria.hal.science/hal-01011633v1

Submitted on 24 Jun 2014

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.

Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures

Luka Stanisic[1], Samuel Thibault[2], Arnaud Legrand[1], Brice Videau[1], and Jean-François Méhaut[1]

[1] CNRS/Inria/University of Grenoble, France, firstname.lastname@imag.fr
[2] University of Bordeaux/Inria, France, samuel.thibault@labri.fr

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 rely 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 for distributed systems. This approach allows to obtain performance predictions accurate within a few percents on classical dense linear algebra kernels 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.

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 [3,4,7]. 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 [3]. 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 tradeoff must be found between large tasks which are efficient on GPUs but expose little task parallelism, and a lot of small tasks for CPUs but are less efficient on GPUs. 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 on 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.

Getting accurate measurement results for all combinations is not trivial and it 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 strategy choices are generic, and not only suited to the few target systems which were available to developers. Finally, since execution time on real machine exhibit variability, dynamic schedulers tend to make varying scheduling decisions, and the obtained performance is thus far from deterministic. This makes performance comparisons more questionable and debugging of non-deterministic deadlocks inside such runtimes even harder.

Simulation is a technique that has proven extremely useful to study complex systems and which would 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 [3] (see Section 3), a dynamic runtime system for heterogeneous multi-core architectures, on top of SimGrid, 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. Our contribution are the following:

- we present in details models that are essential for good performances 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 four different generations of GPUs, within a few seconds on a commodity laptop, and we illustrate how it allows to conduct preliminary exploratory studies easily (Section 8).

2 Related Work

In most other 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 [5], 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 [5] 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 [11] for the AMD Evergreen GPU). Such specialization allow 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 $90,000\times$[11].

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 errors come from an inadequate level of details and can be avoided. Therefore, we propose to use a top-down modeling approach such as promoted by the SimGrid project [8], 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 as a reasonable alternative to both simple analytic models and expensive, difficult-to-instantiate packet-level simulations [12] and have recently been extended to simulate accurately MPI applications on Ethernet networks [6]. 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) [12] at a very low simulation cost. In the

next sections, we explain how StarPU has been ported on top of SimGrid and how multi-GPU 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 StarPU scheduling is generally dynamic and opportunistic, the decisions taken when simulating should be as close as possible to the ones taken in a native execution. 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 had to be abstracted as well. 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. Last, since schedulers may use runtime statistics to take scheduling decisions, time had to be abstracted as well to make sure that simulation time (instead of current time) is used in a consistent way. 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.

For simplicity reasons, each CPU 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 GPUs 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 GPU internals.

In such a modeling, the overhead of the runtime (e.g., the time needed to take scheduling decisions, to manage synchronizations or to manage internal queues) is not accounted for in the simulation and only the parts related to the application execution are simulated. As we will see in the rest of the article,

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
conan	Intel Xeon E5-2650	2×8	2.0GHz	2×32GB	3×TeslaM2075
frogkepler	Intel Xeon E5-2670	2×8	2.6GHz	32GB	2×K20

such a naive 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 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 [2]. 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 that the application can use only computation kernels that have been measured, of course. 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. These NVIDIA GPUs have distinct characteristics and belong to different generations, which intends to demonstrate the validity of our approach on a range of diverse machines. Regarding applications, we decided to focus on two common dense linear algebra kernels: *cholesky* and *LU* factorization. Regarding task granularity, we fixed a relatively large block size (960×960) as it is

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

representative of what is typically used to achieve good performances. In our experiments, CPUs were only controlling the execution and scheduling of the tasks while GPUs had the roles of workers, meaning that 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. We also investigated situations involving both CPUs and GPUs a the same time. Although the initial results were excellent, we could not include them in this article due to lack of room and decided to instead present in detail the specifics of GPU modeling.

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 experiment results presented in this paper are publicly available on *figshare* [13]. 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 allow to regenerate this document.

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). Another (probably the most) influential parameter for accurate modeling

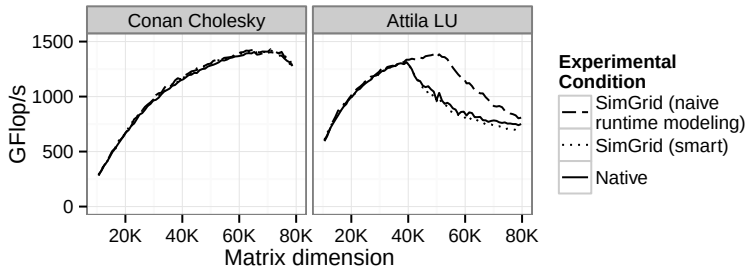


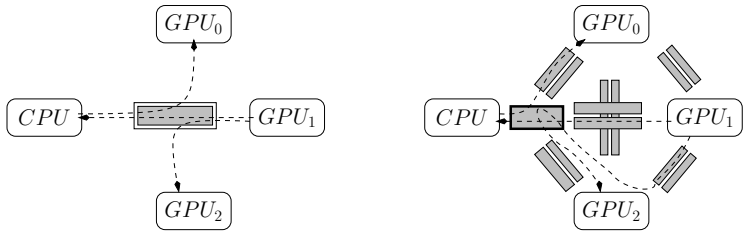
Fig. 1. Illustrating the influence of modeling runtime. Careless modeling of runtime may be perfectly harmless in some cases, it turns out to be misleading in others

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 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 1 illustrates the importance of taking into account the runtime parameters described above. Each curve depicts GFlop/s rate of experiments representing 90 different matrix dimensions (matrix dimension 80,000 corresponds to ≈25GB). Solid line *Native* 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 machines and applications (plot on the right), having a precise modeling of runtime is critical as otherwise, simulation may highly overestimate the performance for the larger matrix size. 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 GPUs. Modeling communication between computing resources is thus of primary importance. As a first approximation (see Figure 2(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 2(b)). Furthermore,



(a) Crude modeling (b) More elaborated modeling

Fig. 2. 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

any communication between two resources has to go through a common shared link (in bold), which represent the maximum capacity of the PCI bus. Modeling contention in such a way is however insufficient as 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.

Additionally, to move chunks of matrices between resources, StarPU relies on the `cudaMemcpy2D` function. First, the performance of this function is not exactly the same as the one of `cudaMemcpy`, which was used in the original calibration process. Even more importantly, 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 3) whereas it can be relatively safely ignored on others. Therefore, communication time is modeled as a piece-wise linear function of data payload and whose slope and intercept depend on the pitch of the matrix.

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 4, a naive network modeling such as the one on Figure 2(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 2(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.

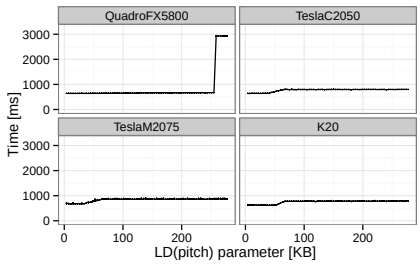


Fig. 3. Transfer time of 3,600 KB using `cudaMemcpy2D` depending on the pitch of the matrix.

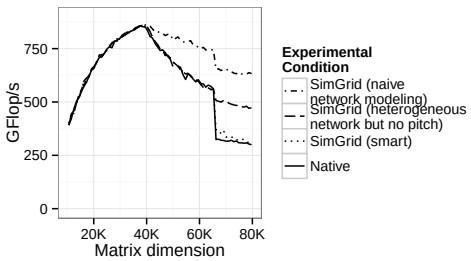


Fig. 4. Performance of the LU application on hannibal (QuadroFX5800 GPUs) using different modeling assumptions

7 Modeling computation

When running 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. In our initial approach, we used the mean duration of each computation kernel, which was benchmarked by StarPU during the calibration phase. Although this was producing satisfactory results, using a fixed value leads to a deterministic schedule in simulation. This may bias the simulation and does not allow to verify the ability of the scheduling algorithms to handle the variability of the resources.

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 [10], as 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 simulation, 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.

8 Prediction Accuracy in a Wide Range of Settings

As we explained in the previous section, 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. Figure 5 depicts the performance as a function of the size of the matrix for the two applications LU and Cholesky and for the four different hybrid systems we described in Table 1. For most combinations, the prediction obtained with SimGrid is very accurate. The only two scenarios where the error is larger than a few percents is for the LU kernel on

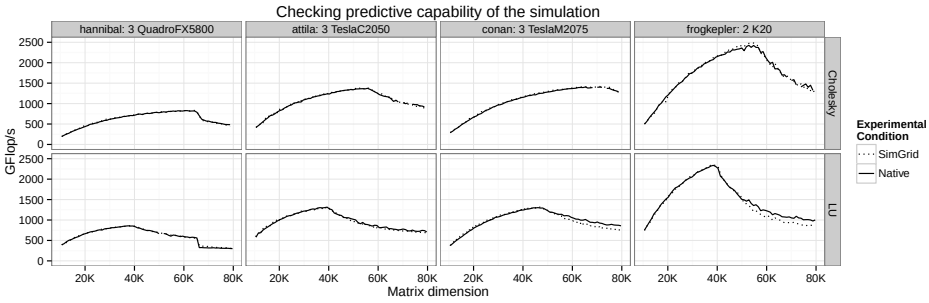


Fig. 5. Checking predictive capability of our simulator in a wide range of settings

conan and frogkepler when our prediction slightly overestimates the (bad) performances for large matrices. The trend is however perfectly predicted as well as the size beyond which performance drops.

A closer look at traces (see Figure 6) 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. Since even with the same parameters, native traces differ from an execution to another, a point-to-point comparison with a simulation trace would not make sense. However, we can check that both traces are indeed extremely close, which allows to study and understand the potential weaknesses of a scheduler.

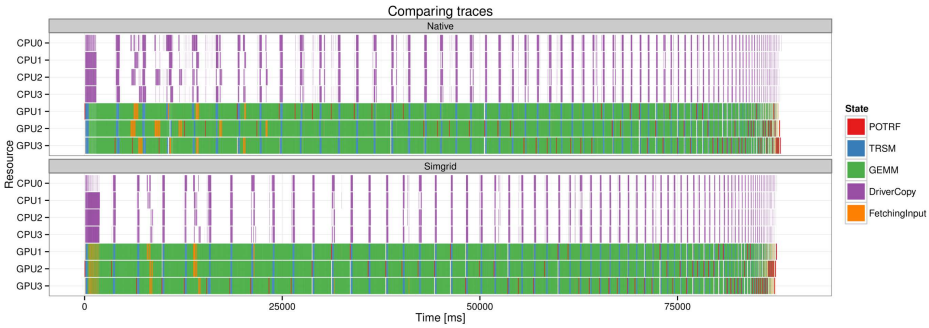


Fig. 6. Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $72,000 \times 72,000$ matrix on the Conan machine. Traces are not perfectly identical since the execution is not deterministic but the behavior of the simulation is representative of the real execution

For example, the reason for the performance drop observed on Figure 5 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 5 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 GPU may be overloaded with work and forced to evict some data, as it cannot handle the whole matrix. Two other strategies *DMDAR* and *DMDAS* were designed to tend to execute 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 7). Later, when the target system became accessible again, we confirmed these results by running the same experiments and as can be seen on Figure 7, our simulations were again perfectly accurate.

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

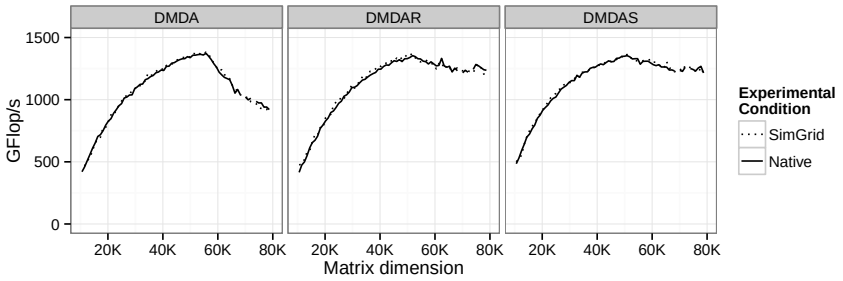


Fig. 7. Cholesky on Attila: studying the impact of different schedulers

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, four different generations of GPUs, several scheduling algorithms) until we found a situation where our simulation failed 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 three directions. First, we plan to explore using both CPUs and GPUs as computation units. While initial investigation on classical hybrid multi-core computers showed perfect results, we expect that dealing with large NUMA machines comprising hundreds of cores will be much harder. Second, StarPU was recently extended to exploit clusters of hybrid machines by relying on MPI [1]. Since SimGrid's ability to accurately simulate MPI applications has already been demonstrated [6], combining both works should allow to obtain good performances predictions of complex applications on large-scale high-end HPC infrastructures. Third, many numerical applications have been recently ported on top of StarPU, including dense (MAGMA and PLASMA) and sparse linear algebra (QR-MUMPS), 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.

Acknowledgments. 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 and Emmanuel Agullo for motivating this study and providing insights on its usefulness.

References

1. Augonnet, C., Aumage, O., Furmento, N., Namyst, R., Thibault, S.: StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In: Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface (EuroMPI). pp. 298–299. Springer-Verlag (2012)
2. Augonnet, C., Thibault, S., Namyst, R.: Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In: 3rd Workshop on Highly Parallel Processing on a Chip (HPPC) (Aug 2009)
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience 23, 187–198 (Feb 2011)
4. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Proceedings of the 15th Euro-Par Conference (Aug 2009)
5. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: ISPASS. pp. 163–174 (2009)
6. 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. In: 4th International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS) (Nov 2013)
7. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A Generic Distributed DAG Engine for High Performance Computing. In: IEEE International Symposium on Parallel and Distributed Processing. pp. 1151–1158. IEEE Computer Society (2011)
8. Casanova, H., Legrand, A., Quinson, M.: SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In: proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation (UKSim) (Apr 2008)
9. Collange, S., Daumas, M., Defour, D., Parello, D.: Barra: A Parallel Functional Simulator for GPGPU. In: IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication. pp. 351–360 (2010)
10. Denby, L., Mallows, C.: Variations on the histogram. Journal of Computational and Graphical Statistics 18(1), 21–31 (2009)
11. Ubal, R., Jang, B., Mistry, P., Schaa, D., Kaeli, D.: Multi2Sim: A Simulation Framework for CPU-GPU Computing. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. pp. 335–344. PACT '12, ACM, New York, NY, USA (2012)
12. 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 23(3) (Oct 2013)
13. Companion of the StarPU+SimGrid article. Hosted on Figshare. (2014), http://dx.doi.org/10.6084/m9.figshare.928095, online version of this article with access to the experimental data and scripts (in the org source).