Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures

Cédric Augonnet, Samuel Thibault, Raymond Namyst

Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures

Cédric Augonnet, Samuel Thibault, and Raymond Namyst

INRIA Bordeaux, LaBRI, University of Bordeaux

Abstract. Multicore architectures featuring specialized accelerators are getting an increasing amount of attention, and this success will probably influence the design of future High Performance Computing hardware. Unfortunately, programmers are actually having a hard time trying to exploit all these heterogeneous computing units efficiently, and most existing efforts simply focus on providing tools to offload some computations on available accelerators. Recently, some runtime systems have been designed that exploit the idea of scheduling – as opposed to offloading – parallel tasks over the whole set of heterogeneous computing units. Scheduling tasks over heterogeneous platforms makes it necessary to use accurate prediction models in order to assign each task to its most adequate computing unit [2]. A deep knowledge of the application is usually required to model per-task performance models, based on the algorithmic complexity of the underlying numeric kernel.
We present an alternate, auto-tuning performance prediction approach based on performance history tables dynamically built during the application run. This approach does not require that the programmer provides some specific information. We show that, thanks to the use of a carefully chosen hash-function, our approach quickly achieves accurate performance estimations automatically. Our approach even outperforms regular *algorithmic* performance models with several linear algebra numerical kernels.

1 Introduction

Multicore architectures are now widely adopted throughout the computer ecosystem. There is also clear evidence that solutions based on specialized hardware, such as accelerator devices (*e.g.* GPGPUs) or integrated coprocessors (*e.g.* Cell's SPUs) are offering promising answers to the physical limits met by processor designers. Future processors will therefore not only get more cores, but some of them will be tailored for specific workloads.

In spite of their promising performance in terms of computational capabilities and power efficiency, such heterogeneous multicore architectures require appropriate tools. This introduces challenging problems at all levels, ranging from programming models and compilers to the design of libraries with a real support for heterogeneity. As they offer dynamic support for what has become hardly doable in a static fashion, runtime systems have a central role in this software

stack. In previous work, we have therefore developed StarPU [2], a unified runtime system that offers support for heterogeneous multicore architectures. Its specificity is that it not only targets accelerators (GPUs, Cell's SPUs, *etc.*) but also multicore processors *at the same time*, in a portable fashion. StarPU also provides portable performance thanks to a high-level framework for designing portable scheduling policies.

Performance modeling is a very common technique in the scheduling literature. Whenever doable, practically building such models usually requires consequent efforts along with a certain knowledge of both the application algorithm and the underlying architecture. This is even more difficult in the case of heterogeneous platforms. But without an appropriate interface, such knowledge is not available from the runtime system's perspective: describing a task as a function pointer and pointers to the data (similar to OpenMP 3.0 tasks) does not really give much information to the runtime system in charge of the scheduling. (Un)fortunately, current accelerators reintroduce the problem of data management across a distributed memory model, so that we *have* to adopt much more expressive task APIs anyway. The majority of the programming models that target accelerators (and that do not just delegate data movements to the programmer!) require to explicitly describe which data is accessed by a task [6,7,10,1]. While this adds constraints on the programmer who has to adapt its applications to those expressive programming interfaces, the underlying runtime system gets much more information.

In this paper, we explain how StarPU takes advantage of that expressiveness to seamlessly build performance models on heterogeneous multicore architectures. Then, we illustrate how this systematic approach performs in terms of prediction accuracy and regarding its impact on the actual performance. Finally, we show that StarPU not only grabs information from the programming interface to perform better scheduling, but it also returns performance feedback information thanks to convenient tools which are helpful for instance in the context of auto-tuned libraries or when analyzing performance.

2 StarPU, a runtime system for heterogeneous machines

In this section, we briefly present StarPU, our unified runtime system designed for heterogeneous multicore platforms, described in more details in a previous paper [2]. It distributes tasks onto both accelerators and processors *simultaneously* while offering portable performance thanks to generic scheduling facilities.

2.1 A unified runtime system

The design of StarPU is organized around three main components: a portable offloable-task abstraction, a library that manages data movements across heterogeneous platforms, and a flexible framework to design portable scheduling policies.

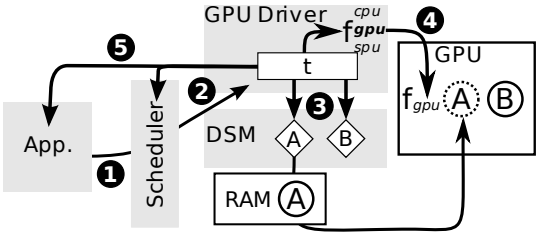

Fig. 1. Execution of a Task within StarPU. Applications submit tasks that are dispatched onto the different drivers by the scheduler. The driver offloads the computation, using the proper implementation from the codelet, and the DSM (Distribution Shared Memory) ensures the availability of coherent data. A callback is executed when the task is done.

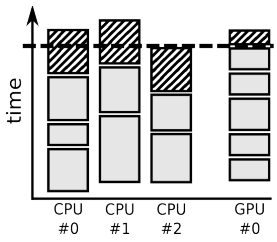

Fig. 2. The "Earliest Finish" Scheduling Strategy.

A unified execution model. StarPU exposes the structure of *codelet*, which is the set of implementations of the same computation kernel (*e.g.* a vector sum) for different computation units (*e.g.* CPU and GPU). A StarPU task is then an instance of a codelet applied to some data. Figure 1 shows the path followed by tasks in StarPU. The programmer explicitly submits (graphs of) tasks to StarPU which maps them as efficiently as possible on the eligible processing units. Instead of hard-coding all the interactions between the processing units, StarPU makes it possible to concentrate on the design of efficient computational kernels and algorithmic problems instead of being stuck by low-level concerns.

A data management library. Maintaining data coherency (and availability) is a crucial issue with accelerators. In a previous paper [1], we have designed a high-level data management library that is integrated in StarPU. Mapping data statically is not necessarily sufficient when multiple processing units access the same pieces of data. The resulting data transfers are critical for the overall performance so that integrating data management within StarPU made it possible to apply optimizations (*e.g.* prefetching, reordering, asynchronous memory transfers) and to guide the scheduler.

A scheduling framework. StarPU not only executes tasks, but it also maps them as efficiently as possible thanks to its expressive scheduling interface. Hence, StarPU offers a flexible framework to implement portable scheduling policies [2]. Such policies are portable in the sense that they are directly applicable to platforms as different as a Cell processor and a hybrid GPU/CPUs machine.

2.2 Scheduling strategies based on performance models

In a previous paper [2], we have presented various scheduling policies implemented in StarPU with relatively little effort. For instance, one of these policies is similar to the HEFT scheduling strategy [12]. As shown in Figure 2, the scheduler keeps track of the expected duration until the different processing units are

available. When a task is submitted to the scheduler, it is attributed to the processing unit that minimizes termination time according to the expected duration of the task on the different architectures (depicted by hatchings).

We have for instance used this rather simple strategy successfully to obtain superlinear speedups on an LU decomposition thanks to per-architecture performance models that take into account the (lack of) affinity of tasks with the different processing units. However this strategy requires that we can approximate the execution time of the tasks on the various architectures.

3 Dynamically building Performance Models

In this section, we discuss how we can build performance models, and we give a systematic approach to dynamically construct and query a performance model based on historical knowledge, seamlessly for the programmer.

In the context of dynamic task scheduling, we do not need perfectly accurate models, but we need to take appropriate decisions when assigning the tasks onto the different processing units. Our performance models should for instance capture the relative speedups as well as the affinities between tasks and processors.

3.1 How to define a performance model?

In order to define a performance model, we need to decide which parameters the model should depend on.

The most obvious parameters to describe a task are the kernel and the architecture: in the case of a matrix product on CUDA, we could for instance identify a task by the pair (SGEMM, CUDA) and associate it with its predicted execution time. A trivial refinement is to consider the total size of the tasks' data, so we can also associate this pair with a parametric cost function depending on that size (*e.g.* $\mathcal{O}\left(S^{3/2}\right)$ in the case of SGEMM applied on a matrix of size S).

The total size is often not sufficient: in the case of a kernel handling a $(n \times m)$ matrix in $\mathcal{O}(n^2 m)$, we must make a distinction between (1024×512) and (512×1024) matrices (for example). Such multivariate models are however only applicable if we have sufficient knowledge of the algorithm, which a runtime system could hardly infer automatically in a generic way. Finding an explicit model of the execution time can also be awkward because of architectural concerns such as the size of caches. Using piecewise models is possible, but it requires to delimit the boundaries of the pieces, which can be time demanding, especially for a multivariate model and in a heterogeneous environment.

In many classes of algorithms, we can reasonably make some extra regularity assumption such as most tasks handling blocks of fixed size (*e.g.* in tiled algorithms), or a limited set of sizes (*e.g.* in divide and conquer algorithms). In this case, explicitly modeling the performance as a function of the data size can be unnecessarily complicated. A history-based approach would be much simpler: instead of using a complicated multivariate model to differentiate between a (1024×512) matrix and a (512×1024) one, we simply store the execution

time that was measured for those different input configurations. The advantage of this approach is that it is transparent to the programmer as long we have some mechanism to match a task with those previously executed. This method is however not applicable to irregular applications: if we know the performance of a kernel on a (1024×512) matrix, no prediction can be made for a (1026×510) matrix for instance. In the next section, we present how StarPU implements history-based models with sufficient performance feedback to help programmer easily decide whether this is an appropriate model or not.

3.2 How to build performance models?

There are various ways to determine the parameters required to build the performance models that we have described in the previous section: either completely manual, or completely automated, depending on the type of the adopted model.

Building a performance model by hand (*e.g.* using the ratio between the number of operations and the speed of the processor) is hardly applicable to modern processors and require a detailed knowledge of both the application and the architecture. In the case of heterogeneous multicore processors, with multiple processing units to handle, this becomes rather unrealistic. It is however possible to design a model based on the amount of computations per task, and to calibrate the parameters by the means of a regression.

It is common to use specific precalibration programs to build those models. While this may be suited for kernels that are widely used (*e.g.* BLAS), this requires a specific test-suite and the corresponding inputs, which often represents an important programming overhead. In the context of multicore architectures, it is even harder to create a realistic workload: independently benchmarking the various processing units without taking into account the various interactions (*e.g.* cache sharing or bus contention) may not result in reliable measures.

On the other hand, it is possible to measure the performance of the different tasks during an actual execution. This does not require any additional programs, and it provides realistic performance measurements. StarPU can therefore automatically calibrate parametric models, either at runtime using linear regression models (*e.g.* of the form $\mathcal{O}(n^\alpha)$) or offline in the case of non-linear models (*e.g.* of the form $\alpha n^\beta + \gamma$, as shown in Figure 7). StarPU also builds history-based performance models by storing the performance of the tasks on different inputs, transparently for the application.

3.3 A generic approach for building history-based performance models dynamically

This section shows how StarPU keeps track of the performance obtained by the tasks on the different input, and how it is possible to match a task with its similar predecessors. As shown in Figure 3, this process involves three main steps: measuring the actual duration of the tasks when they are executed and

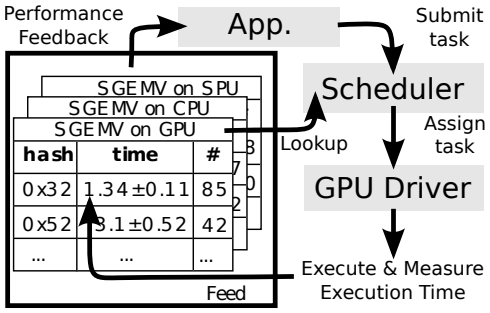

Fig. 3. Performance feedback loop.

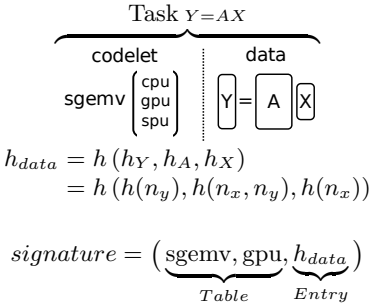

Fig. 4. Uniquely identifying a task

integrating these measurements in the history log of the task; being able to look-up the performance of some task according to the previous measurements; and offering some performance feedback to the application.

Measuring tasks' duration. Measuring the time spent to compute a task is usually simple thanks to the cycle counter facility provided by most manufacturers. In the case of Cell processors, which lacks such functionality, we had to make the SPUs transmit those measurements to the PPU along with the output data, this is not intrusive since DMA transfers are overlapped.

Identifying task kinds. We use the layout and size of the data to distinguish the different kind of instances of a computational kernel. We now explain how to compute a hash value to characterize the data layout of a task.

StarPU's data management library not only manipulates buffers described by a pointer and its length, but it also handles a mixture of various high-level data *interfaces* [1]. In Figure 4, a matrix-vector product accesses a set of matrices and vectors. There can also be much more complex data interfaces (*e.g.* compressed sparse matrices), but the size of any piece of data must be characterized by a k-tuple of parameters $(p_1, \ldots, p_k)$ where k and the parameters depend only on the data interface. A matrix is for instance described by a pair (n, m), and a single parameter is sufficient to describe the length of a vector.

We now define a hash function that computes a unique identifier for such a set of parameters. As shown in Figure 4, we characterize the size of each piece of data by applying a hash function [1] to the parameters $p_1, \ldots, p_{k-1}$ describing it. By then applying the hash function to the different per-data hashes, we get a characterization of the data layout and size for the whole task. Applying this method on a tiled algorithm would for instance result in having as many hash values as there are tile sizes.

Feeding and looking up from the model. It is now extremely simple to implement a model based on the history in StarPU: each computational kernel is associated with a hash table per architecture. When a task is submitted to

[1] For example, we can use the usual CRC hash functions: $h(p_1, \ldots, p_k) = CRC(p_1, \ldots, CRC(p_{k-1}, CRC(p_k, 0)))$.

StarPU, it computes its hash, and consults the hash table corresponding to the proper kernel-architecture pair to retrieve the average execution time previously measured for this kind of task. The average execution time and other metrics such as standard deviation are updated when a new measure is available. Hash tables can be saved (or loaded) to (from) a file so that these performance models are persistent between different runs. We therefore rapidly calibrate models by running small problems that have the same granularity as the actual problems.

4 Experimental validation

We have implemented these automatic model calibration mechanisms in StarPU which runs on multicore CPUs, GPUs and Cell processors. In this section, we give evidence that they have a significant impact on performance; we also illustrate the performance feedback offered by StarPU, and how StarPU provides some tools to help the programmer to understand the obtained performance, and to select the most appropriate models in consequence. We here show how these mechanisms perform in the case of a hybrid platform with a nVidia Quadro FX4600 GPU and a E5410 Xeon quad-core CPU.

4.1 Sharpness of the performance prediction

Figure 5 shows the results obtained on an LU decomposition for two different problem sizes. The first line exhibits the average and standard deviation of the reference performance obtained when using a greedy scheduling policy to distribute tasks to CPUs and the GPU. The second line shows the results obtained when calibrating the history-based performance model after either one, two or three runs and the average performance (and standard deviation) obtained after 4 runs. During the first execution, the greedy strategy clearly outperforms the non-calibrated strategy based on performance models. But once the model is calibrated, the performance obtained by the model-based strategy gets better, not only in terms of average speed, but also with respect to the standard deviation. The improvement between the runs is explained by the fact that the application

		Speed (GFlop/s)	
Policy \ Size		$(16k \times 16k)$	$(30k \times 30k)$
Greedy (avg.)		89.98 ± 2.97	130.68 ± 1.66
Perf. Model	1^{st} iter.	48.31	96.63
	2^{nd} iter.	103.62	130.23
	3^{rd} iter.	103.11	133.50
	≥ 4 (avg.)	103.92 ± 0.46	135.90 ± 0.64

Fig. 5. Impact of performance sampling on the speed of an LU decomposition (in GFlop/s)

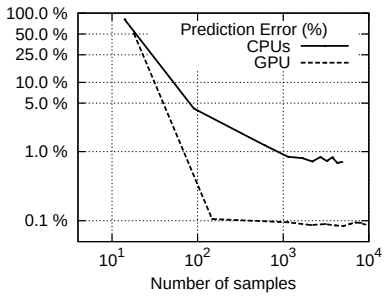

Fig. 6. Performance model accuracy

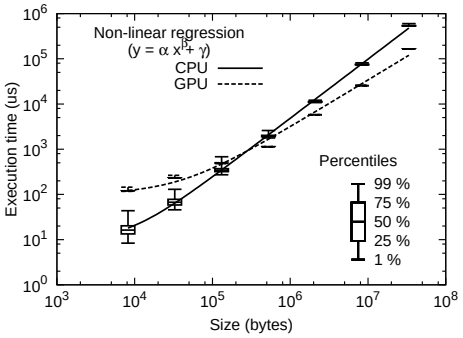

Fig. 7. Performance and regularity of an STRSM BLAS3 kernel depending on granularity.

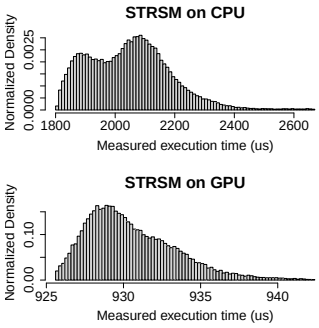

Fig. 8. Distribution of the execution times of a STRSM kernel measured for tiles of size (512×512).

runs on a hybrid CPU/GPU platform: the better the accuracy, the better the load balancing. Until the models are properly calibrated, some processing units receive too much work while others are not kept busy enough.

Figure 6 depicts the evolution of the prediction inaccuracies depending on the number of collected samples. More precisely, the error is computed by taking the sum of the absolute differences between prediction and measurements, for all tasks, and by dividing this total prediction error by the total execution time. As suggested by Figure 5, the accuracy of the models becomes better as we keep collecting measurements. We finally obtain an accuracy of an order of 1% for multicore CPUs, and below 0.1% for a GPU. This difference is due to complex interactions occuring within multicore CPUs (*e.g.* cache sharing and contention) while computations are not perturbed on GPUs. The large majority of tasks in an LU decomposition are matrix products, whose performance is especially regular even on CPUs, so that we obtain a relatively good overall accuracy.

4.2 Performance feedback tools

StarPU provides tools to detect tasks that are not predictable enough (*e.g.* BLAS1 kernels). Figures 7 and 8 are automatically generated by StarPU, which can collect performance measurements at runtime.

Figure 7 summarizes the behaviour of a kernel on all input sizes and the performance variations observed for the different sizes, and for the different architectures; it also shows the non-linear regression-based performance models automatically generated by StarPU so that we can figure out whether such a model is applicable or not. It also illustrates in which situation it is worth using accelerators or CPUs, therefore helping to select the most appropriate granularity. Using a small grain size on CPUs results in variable execution times, certainly explained by a poor cache use which makes performance very sensitive

to the bus contention for instance. This problem disappears as we take large tiles, or if we use a GPU that is much less sensitive to such variations.

Figure 8 shows the actual distribution of the measurements that were collected for a given hash value. This not only gives a precise idea of the performance dispersion, but it can also be used to understand the actual performance issues: on the very predictable GPUs, we obtain a very thin peak, while on the CPUs, the distribution exhibiting two hills suggests that there may be some contention issue which should be further analyzed.

5 Related Works

Auto-tuning techniques have been successfully used to automatically generate the kernels of various high-performance libraries such as ATLAS [4], FFTW, OSKI or SPIRAL; and similar results are obtained in the context of GPU computing by the MAGMA project[9]. While performance models permit to generate efficient computational kernels even on heterogeneous systems, computations are usually mapped statically on the different processing resources when dealing with hybrid systems [11].

Iterative compilation frameworks also use performance feedback to take the most appropriate optimization decisions. Jimenez *et al.* [8] keep track of the relative speedups of the applications on the different architectures to decide which processing unit should be assigned to an application. Their approach is much less flexible since it does not allow to actually schedule interdependent tasks within an application.

Different runtime systems currently offer support for accelerators [3], or even hybrid systems. Similarly to StarPU, the Harmony runtime system targets hybrid platforms while proposing some scheduling facilities, possibly based on performance modeling [5]. Its performance is modeled by the means of (possibly multivariate) regression models. This approach is hardly applicable without any support from the programmer, and possibly requires a large number of samples to have a reliable model. Thanks to the high-level support for data management integrated within StarPU, the history-based solution that we propose in this paper is simpler as it is completely transparent for the programmer.

6 Conclusion

We have proposed a generic approach to seamlessly build history-based performance models. It has been implemented within the StarPU runtime system with the support of its integrated data management library, and we have shown how StarPU's performance feedback tools help the programmer to analyze whether the resulting performance prediction are relevant or not.

Such history-based performance models naturally rely on some regularity hypothesis since it cannot predict the behaviour of a task if all its predecessors had different sizes: in that case, a parametric performance model calibrated by the means of regressions is more suitable. Our history-based approach also

requires computational kernel with a static flow control. Tasks' execution time should be independent from the actual content of the data, the latter is often unknown when the scheduling decisions are taken anyway. This does not require any effort from the programmer who can easily use our auto-tuning mechanisms to see whether such models results into performance improvements or not.

This technique is directly applicable to the case of complex hybrid setups (*e.g.* heterogeneous multi-GPU). This work could also be extended to model the performance of memory transfers to schedule them as well. Scheduling policies could take advantage of performance models that depend on the actual state of the underlying machine: using hardware performance counters, the history-based models could for instance keep track of contention or cache usage. Finally, performance feedback can be valuable: this not only helps to understand the behaviour of an application during a post-mortem analysis, but this is also useful for iterative compilation environments and auto-tuned libraries.

References

1. C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multicore architectures. In *Euro-Par 2008 Workshops - HPPC'08*, Las Palmas de Gran Canaria, Spain, August 2008.
2. C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th Euro-Par Conference*, Delft, The Netherlands, August 2009.
3. P. Bellens, J.M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *Proceedings of SC'06, Tampa, Florida*, 2006.
4. R. Clint Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of SIAM PP'99, San Antonio, Texas*, March 1999.
5. G. Diamos and S. Yalamanchili. Harmony: Runtime Techniques for Dynamic Concurrency Inference, Resource Constrained Hierarchical Scheduling, and Online Optimization in Heterogeneous Multiprocessor Systems. Technical report, Georgia Institute of Technology, Computer Architecture and Systems Lab, 2008.
6. A. Duran, J.M. Perez, E. Ayguade, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependant tasks. In *IWOMP Proceedings*, 2008.
7. K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D. Reiter Horn, L. Leem, J. Young Park, M. Ren, A. Aiken, W.J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of SC'06, Tampa, Florida*, 2006.
8. V.J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of HiPEAC'09, Paphos, Cyprus*, 2009.
9. Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *Proceeding of ICCS'09, Baton Rouge, Louisiana, U.S.A.*, 2009.
10. M.D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx'06 Multicore Applications Conference*, 2006.
11. S. Tomov, J. Dongarra, and M. Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. Technical report, January 2009.
12. H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar 2002.