StarPU: a unified platform for task scheduling on heterogeneous multicore architectures

Cédric Augonnet[*,†], Samuel Thibault, Raymond Namyst and Pierre-André Wacrenier

University of Bordeaux, LaBRI–INRIA Bordeaux Sud-Ouest, Talence, France

SUMMARY

In the field of HPC, the current hardware trend is to design multiprocessor architectures featuring heterogeneous technologies such as specialized coprocessors (e.g. Cell/BE) or data-parallel accelerators (e.g. GPUs). Approaching the theoretical performance of these architectures is a complex issue. Indeed, substantial efforts have already been devoted to efficiently offload parts of the computations. However, designing an execution model that unifies all computing units and associated embedded memory remains a main challenge. We therefore designed StarPU, an original runtime system providing a high-level, unified execution model tightly coupled with an expressive data management library. The main goal of StarPU is to provide numerical kernel designers with a convenient way to generate parallel tasks over heterogeneous hardware on the one hand, and easily develop and tune powerful scheduling algorithms on the other hand. We have developed several strategies that can be selected seamlessly at run-time, and we have analyzed their efficiency on several algorithms running simultaneously over multiple cores and a GPU. In addition to substantial improvements regarding execution times, we have obtained consistent *superlinear* parallelism by actually *exploiting* the heterogeneous nature of the machine. We eventually show that our dynamic approach competes with the highly optimized MAGMA library and overcomes the limitations of the corresponding static scheduling in a portable way. Copyright © 2010 John Wiley & Sons, Ltd.

1. INTRODUCTION

Multicore processors are now mainstream. To face the ever-increasing demand for more computational power, HPC architectures are not only going to be *massively* multicore, they are going to feature *heterogeneous* technologies such as specialized coprocessors (e.g. Cell/BE SPUs) or data-parallel accelerators (e.g. GPGPUs). As illustrated by the currently Top500-leading IBM Road-Runner machine, which is composed of a mix of Cells and Opterons, accelerators have indeed gained a significant audience. In fact, heterogeneity not only affects the design of computing units itself (CPU vs Cell's SPU), but also the memory design (cache and memory banks hierarchy) and even programming paradigms (MIMD vs SIMD).

But is HPC, and its usual momentum, ready to face such a revolution? At the moment, this is clearly not the case, and progress will only come from our ability to harness such architectures while requiring minimal changes to programmers' habits. As none of the main programming standards

[*]Correspondence to: Cédric Augonnet, University of Bordeaux, LaBRI–INRIA Bordeaux Sud-Ouest, Talence, France.
[†]E-mail: cedric.augonnet@inria.fr

(i.e. MPI and OPENMP) currently address all the requirements of heterogeneous machines, important standardization efforts are needed. Unless such efforts are made, accelerators *will* remain a niche. In this respect, the OpenCL initiative is clearly a valuable attempt in providing a common programming interface for CPUs, GPGPUs, and possibly other accelerators. However, the OpenCL API is a very low-level one which basically offers primitives for explicitly offloading tasks or moving data between coprocessors. It provides no support for task scheduling or global data consistency, and thus cannot be considered as a true 'runtime system', but rather as a virtual device driver.

To bridge the gap between such APIs and HPC applications, one crucial step is to provide optimized versions of computation kernels (BLAS routines, FFT, etc.) capable of running seamlessly over heterogeneous architectures. These kernels must be designed to dynamically adapt themselves to the available resources and application load because due to heterogeneity, generic static mapping strategies are unlikely to emerge in the near future.

As an attempt to provide a runtime system allowing the implementation of such numerical kernels, we have recently developed a data-management library that seamlessly enforces a coherent view between, e.g., main memory and GPU memory [1]. This library was successfully used to implement some simple numerical kernels quite easily the next step is now to abstract the concept of task on heterogeneous hardware and to provide expert programmers with scheduling facilities. Integrated within high-level programming environments such as OPENMP, this would indeed help application developers to concentrate on high-level algorithmic issues, regardless of the underlying scheduling issues.

We here propose StarPU, a simple tasking API that provides numerical kernel designers with a convenient way to execute parallel tasks over heterogeneous hardware on the one hand, and easily develop and tune powerful scheduling algorithms on the other hand. StarPU is based on the integration of the data-management facility with a task execution engine. We demonstrate the relevance of our approach by showing how heterogeneous parallel versions of some numerical kernels were developed together with advanced scheduling policies.

Section 2 presents the unified model we propose to design in StarPU. In Section 3, we enrich our model with support for scheduling policies. We study how scheduling improves the performance of our model in Section 4. We compare our results with similar works in Section 5, and Section 6 draws a conclusion and plans for future work.

2. THE STARPU RUNTIME SYSTEM

Each accelerator technology usually has its specific execution model (e.g. CUDA for NVIDIA GPUs), and its proper interface to manipulate data (e.g. DMA on the CELL). Porting an application to a new platform therefore often boils down to rewriting a large part of the application, which severely impairs productivity. Writing portable code that runs on multiple targets is currently a major issue, especially if the application needs to exploit multiple accelerator technologies, possibly at the same time.

To help tackling this issue, we designed StarPU, a runtime layer that provides an interface unifying execution on accelerator technologies as well as multicore processors. Middle layers tools (such as programming environments and HPC libraries) can build up on top of StarPU (instead of directly using low-level offloading libraries) to keep focused on their specific roles instead of having to handle efficient simultaneous use of offloading libraries. That allows programmers to make existing applications efficiently exploit different accelerators with limited effort. We now present the main components of StarPU: a high-level library that takes care of transparently performing data movements (already described in a previous article [1]), and the new unified execution model.

2.1. Data management

As accelerators and processors usually cannot transparently access the memory of each other, performing computations on such architectures implies explicitly moving data between the various computational units. Considering that multiple technologies may interact, and that specific

knowledge is required to handle the variety of low-level techniques, in previous work [1] we designed a high-level library that efficiently automates data transfers throughout heterogeneous machines. It uses a software MSI caching protocol to minimize the number of transfers, as well as partitioning functions and eviction heuristics to overcome the limited amount of memory available on accelerators.

2.2. An accelerator-friendly unified execution model

The variety of technologies makes accelerator programming highly dependant on the underlying architecture, so we propose a uniform approach for task and data parallelism on heterogeneous platforms. We define an abstraction of a task (e.g. a matrix multiplication) that can be executed on a core or offloaded onto accelerators asynchronously. Programmers can implement the task on multiple targets directly by the means of the programming languages (e.g. CUDA) or with the libraries (e.g. BLAS routines) usually available on those architectures, and the abstraction is the set of all these implementations. From a programming point of view, StarPU is responsible for executing the tasks that were submitted. The application does not have to consider the problem of load balancing and task offloading.

Declaring tasks and data dependencies: The StarPU task structure includes a high-level description of every piece of data manipulated by the task and how they are accessed (i.e. R, W, R/W). It is also possible to express tasks dependencies, so that StarPU not only enforces data coherency thanks to its high-level data management library, but it also allows programmers to express complex task graphs with very little efforts. Since tasks are launched asynchronously, this also allows StarPU to reorder them when that improves performance.

Porting StarPU to various hardware: The task model adopted by StarPU is powerful enough to fit the different (co)processors technologies. While the core of StarPU takes care of handling data movements along scheduling tasks regardless of the underlying hardware, supporting a new architecture only requires little efforts. Such a driver should first provide the functions to transfer memory between the host and the accelerator. It then must implement the method that actually executes (or offload) a task onto the accelerators, typically by the means of the architecture's proprietary interface or with some third-party architecture-specific library. This model was successfully implemented on top of multicore processors, on CUDA-enabled GPUs, and on the CELL processor [2].

3. A GENERIC SCHEDULING FRAMEWORK FOR HETEROGENEOUS ARCHITECTURES

The previous section has shown how tasks can be executed on the various processing units of a heterogeneous machine. However, we did not specify how they should be distributed efficiently, especially with regards to load balancing. It should be noted that nowadays architectures have gotten so complex that it is very unlikely that writing portable code which efficiently maps tasks statically is either possible or even productive.

3.1. Scheduling tasks in a heterogeneous world

Data transfers have an important impact on performance, so that a scheduler favoring locality may increase the benefits of caching techniques by improving data reuse. Considering that multiple problems may be solved concurrently, and that machines are not necessarily fully dedicated (e.g. when coupling codes), dynamic scheduling becomes a necessity. In the context of heterogeneous platforms, the performance vary a lot according to architectures and according to the workload (e.g. SIMD code vs irregular memory access). It is therefore crucial to take the specificity of each computing unit into account when assigning work.

Similar to the problems of data transfers or task offloading, heterogeneity makes the design and the implementation of portable scheduling policies a challenging issue. Hence, we propose to extend our uniform execution model with a uniform interface to design task schedulers. StarPU offers low-level scheduling mechanisms (e.g. work stealing) so that scheduler programmers can use them in a high-level fashion, regardless of the underlying (possibly heterogeneous) target

Table I. Scheduling policies implemented using our interface.

Name	Policy description
greedy	Greedy policy with support for priorities
no-prio	Greedy policy without support for priorities
ws	Greedy policy based on Work Stealing
w-rand	Random weighted by processor speeds
heft-tm	Heterogeneous Earliest Finish Time [3]

architecture. Since all scheduling strategies have to implement the same interface, they can be programmed independently from applications, and the user can select the most appropriate strategy at runtime.

In our model, each **worker** (i.e. each computation resource) is given an *abstract* queue of tasks. Two operations can be performed on that queue: task submission (push), and request for a task to execute (pop). The actual queue may be shared by several workers provided its implementation takes care of protecting it from concurrent accesses, thus making it totally transparent for the drivers. All scheduling decisions are typically made within the context of calls to those functions, but there is nothing that prevents a strategy from being called in other circumstances or even periodically.

In essence, defining a scheduling policy consists in creating a set of queues and associating them with the different workers. Various designs can be used to implement the queues (e.g. FIFOs or stacks), and queues can be organized according to different topologies (e.g. a central queue, or per-worker queues). Differences between strategies typically result from the way one of the queue is chosen when assigning a new task during its submission.

3.2. Writing portable scheduling algorithms

Since they naturally fit our queue-based design, all the strategies that we have written with our interface (Table I) implement a greedy *list scheduling* paradigm: when a ready task (i.e. all its dependencies are fulfilled) is submitted, it is directly inserted in one of the queues, and former scheduling decisions are not reconsidered. Contrary to DAG scheduling policies, we do not schedule tasks that are not yet ready: when the last dependency of a task is executed, StarPU schedules it by the means of a call to the usual push function.

Restricting ourselves to list scheduling may somehow reduce the generality of our scheduling engine, but this paradigm is simple enough to make it possible to implement portable scheduling policies. This is not only transparent for the application, but also for the drivers that request work. This simplicity allows people working in the field of scheduling theory to plug higher-level tools to use StarPU as an experimental playground.

Moreover, preliminary results confirm that using task queues in the scheduling engine is powerful enough to efficiently exploit the specificities of the CELL processor; and the design of OPENCL is also based on task queues: list scheduling with our push and pop operations is a simple, yet expressive paradigm.

3.3. Scheduling hints

To investigate the possible scope of performance improvements thanks to better scheduling, we let the programmer add some extra optional scheduling hints within the task structure. One of our objective is to fill the gap between the tremendous amount of work that has been done in the field of scheduling theory and the need to benefit from those theoretical approaches on actual machines.

Declaring prioritized tasks: The first addition is to let the programmer specify the level of priority of a task. Such priorities typically prevent crucial tasks from having their execution delayed too much. While describing which tasks should be prioritized usually already makes sense from an algorithmic point of view, it could also be possible to infer it from an analysis of the task DAG.

Guiding scheduling policies with performance models: Many theoretical studies of scheduling problems assume having a *weighted* DAG of the tasks [4]. Whenever it is possible, we thus propose to let the programmer specify a performance model to extend the dependency graph with weights. Scheduling policies can subsequently eliminate the source of load imbalance by distributing work with respect to the amount of computation that has already been attributed to the various processing units.

The use of performance models is actually fairly common in high-performance libraries. Various techniques are thus available to allow the programmer to make performance predictions. Some libraries exploit the performance models of computation kernels that have been studied extensively (e.g. BLAS). This for instance makes it possible to select an appropriate granularity [5] or even a better (static) scheduling. It is also possible to use sampling techniques to automatically determine such model costs, provided actual measurements. In StarPU that can be done either by the means of a pre-calibration run, using the results of previous executions, or even by dynamically adapting the model with respect to the running execution.

The heterogeneous nature of the different workers makes performance prediction even more complex. Scheduling theory literature often assumes that there is a mathematical model for the amount of computation (i.e. in FLOP), and that execution time may be computed according to the relative speed of each processor (i.e. in FLOP/S) [4]. However, it is possible that a task is implemented using different algorithms (with different algorithmic complexities) on the various architectures. As the efficiency of an algorithm heavily depends of the underlying architecture, another solution is to create performance models for each architecture. In the following section, we compare both approaches and analyze the impact of model accuracy on performances.

3.4. Predefined scheduling policies

We currently implemented a set of common queue designs (stack, FIFO, priority FIFO, deques) that can be directly manipulated within the different methods in a high-level fashion. The policy can also decide to create different queue topologies, for instance, a central queue or per-worker queues. The push and the pop methods are then responsible for implementing the load balancing strategy. Defining a policy with our model only consists in defining a couple of methods: a method called at the initialization of StarPU, and the push and the pop methods that implement the interaction with the abstract queue. Table I shows a list of scheduling policies that were designed usually in less than 100 lines of C code, which shows the conciseness of our approach. The next section provides more details and evaluates their benefits.

4. SCHEDULING EXPERIMENTATION

To validate our approach, we present several scheduling policies and experiment them in StarPU on a few applications. To investigate the scope of improvements that scheduling can offer, we gradually increase the quality of the hints given to StarPU, either thanks to automatic performance prediction mechanisms, or by the programmer. We then analyze in more details how StarPU takes advantage of scheduling to exploit heterogeneous machines efficiently.

4.1. Experimental testbed

Our experiments were performed on two machines with a E5410 XEON quad-core running at 2.33 GHz with 4 GB of memory. They are equipped with an NVIDIA QUADRO FX4600 graphic card (resp. NVIDIA QUADRO FX5800) with 768 MB of memory (resp. 4 GB). This machine runs LINUX 2.6 and CUDA 2.3. Unless specified otherwise, we used the ATLAS 3.8 and CUBLAS 2.3. Given CUDA current limitations, one core is dedicated to controlling the GPU efficiently [1] so that e.g. on the first machine we compute on three cores and a GPU at the same time.

We have implemented several (single and double precision) numerical algorithms in order to analyze the behavior of StarPU and the impact of scheduling on the performance. *A blocked matrix multiplication* that will help us demonstrate that greedy policies are not always effective, even on
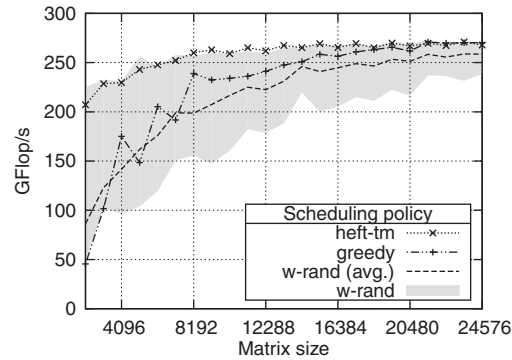
Figure 1. Blocked matrix multiplication.

such simple algorithms. *A blocked Cholesky decomposition* (without pivoting) which emphasizes the need for priority-based scheduling. *A blocked* LU *decomposition* (with and without pivoting) which is similar to Cholesky but performs twice as much computation. This demonstrates how our system tackles load balancing issues while actually taking advantage of heterogeneous platforms.

These algorithms are compute-bound as they mostly involve BLAS3 kernels ($\mathcal{O}(n^3)$ operations against $\mathcal{O}(n^2)$ memory accesses). Performance figures are shown in synthetic GFLOP/s as this gives an evaluation of the efficiency of the computation (speedups are not relevant on such heterogeneous platforms).

4.2. Exploiting performance models

Figure 1 demonstrates that the simple **greedy** policy only delivers only 50 GFLOPs for small matrix multiplications, and around 225 GFLOPs for medium-sized problems. That low efficiency is explained by the important load imbalance. With a greedy approach, CPUs grab tasks even if the GPU is about to become available and execute it much more efficiently. As a result, when the GPU has finished its task, the CPUs still need a long time to finish computing their last tasks.

Intuitively, giving *n* times as many tasks to a GPU as to a CPU if the GPU computes *n* times faster than the CPU solves this issue. The next section shows how that is achieved with performance models.

4.2.1. Average acceleration-based performance model.
In the **w-rand** (*weighted random*) strategy, each worker is associated with a ratio that can be seen as an *acceleration* factor. Each time a task is submitted, one of the workers is selected with a probability proportional to its ratio. That ratio can, for instance, be set by the programmer once for all on the machine, or be measured thanks to reference benchmarks (e.g. BLAS kernels). The **w-rand** strategy is typically suited for independent tasks of equal size.

Still, this policy produces extremely variable schedules: the shaded area in Figure 1 shows the variation of performance obtained by the **w-rand** strategy (after excluding the first and the last decile). The lack of load balancing mechanism explains why the average value is also worse than the *greedy* policy for large problems. This simple strategy however gives interesting improvements on LU and Cholesky decompositions as shown in Figure 2(a), for instance. Similar to the matrix product experiment, this strategy is only effective while load balance is not too affected. This suggests that we not only need a strategy that does take into account heterogeneity, such as **w-rand**, but we also need load balancing mechanisms: this is achieved by the **heft-tm** strategy which we present in the next section.

4.2.2. Per-task accurate performance models.
Evenly distributing tasks over workers with regard to their respective speed do not necessarily make sense for tasks that are not equally expensive, or when there are dependencies between them. Hence, programmers can specify a *cost model* for
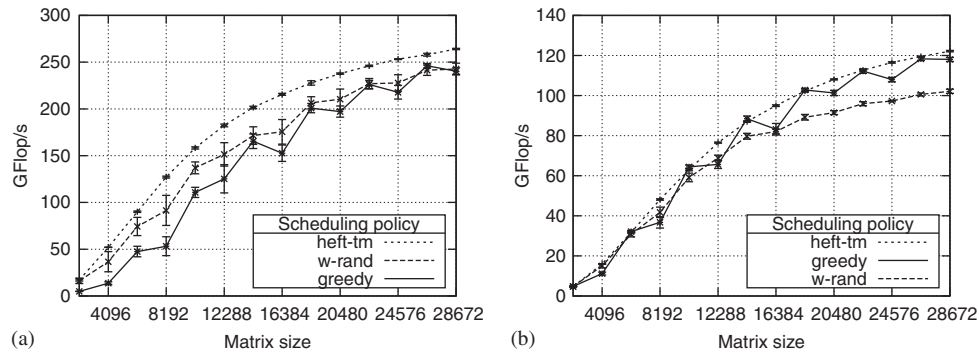
Figure 2. Impact of scheduling policies on single precision LU decomposition: (a) 3 CPUs + Quadro FX5800 and (b) 3 CPUS + Quadro FX4600.

each task. This model can, for instance, represent the amount of work and be used in conjunction with the relative speed of each workers. It can also directly model the execution time on each of the architectures. Using such models, we implemented the Heterogeneous Earliest Finish Time (HEFT) scheduling algorithm [3] in the **heft-tm** strategy. Given its expected duration on the various architectures, a task is assigned to the processing units that minimizes termination time, with respect to the amount of work already attributed to this worker.

By severely reducing load balancing issues, we obtained substantial improvements over our previous strategies, for all benchmarks. Figure 1 shows that even though there is a limited number of independent tasks, we always obtain almost the best execution for matrix multiplication. This strategy improves the performance of medium- and small-sized problems up to twofold, but we observe similar results for large ones because the greedy strategy does not suffer too much load imbalance on large inputs. The **heft-tm** strategy is slightly handicapped by a limited support for prioritized tasks, which is, for instance, useful on the Cholesky benchmark, as will be seen in Section 4.3. Likewise, Figure 2(a) shows that the LU decomposition obtains significant performance improvements as we severely reduce load imbalance, thus reducing execution time by a factor of 2 on small size problems and increasing asymptotic speed. The **heft-tm** strategy is not only faster, it is also much more stable as we consistently observe better performance than with other strategies with a very limited variability. In Figure 2(b), which shows the results with a slower GPU, **heft-tm** also performs the best. In comparison, the **greedy** strategy is almost as fast since the relative speedup of that GPU is limited, thus reducing the risk of load imbalance; it however obtains unstable performance.

In Figure 3, we applied a random perturbation of the performance prediction to show that our scheduling is robust to model inaccuracies. When a task is scheduled, we do not consider its actual predicted length P, but $e^{\alpha \ln(P)}$ with α a random variable selected uniformly from $[1 - \eta; 1 + \eta]$ where η is the perturbation. A 5% perturbation here results in very low performance degradation, and really important mis-predictions are required before *heft-tm* is outperformed by strategies that do not use performance models.

In a previous paper [6], we have shown that StarPU can automatically construct performance models transparently for the programmers. Since the predictions resulting from those automatic performance models are typically within 1% of the actual execution time for regular kernels (e.g. BLAS3 kernels), it is thus possible to get the benefits of the **heft-tm** strategy without any additional effort for the programmers.

4.3. Involving the programmer

Scheduling policies with support for prioritized tasks help reducing load imbalance for algorithms that suffer insufficient parallelism. Table II shows the performance of both Cholesky and LU factorizations with or without priority tasks when using the **heft-tm** scheduling strategy. Priorities have less impact in the case of LU decomposition because it has twice as much computations as the Cholesky algorithm. The improvement that results from priorities is also limited since we consider our
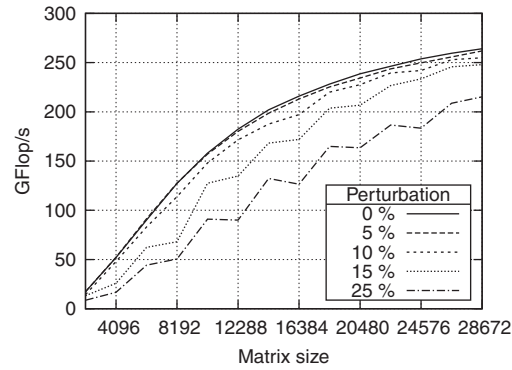
Figure 3. Impact of the model inaccuracies on LU.

Table II. Impact of priorities on the factorization of a 2.3 GB simple precision matrix (in GFlop/s).

Algorithm	Platform	Without priorities	With priorities	Gain (%)
Cholesky	3 CPUs + 1 FX5800	230.55 ± 0.69	238.16 ± 1.38	3.3
	3 CPUs + 1 FX4600	101.13 ± 0.44	104.77 ± 0.47	3.6
LU	3 CPUs + 1 FX5800	252.27 ± 0.23	253.26 ± 0.45	0.5
	3 CPUs + 1 FX4600	110.66 ± 0.33	112.63 ± 0.36	1.8

best scheduling policy, **heft-tm**, whose goal is also to reduce load imbalance. This example illustrates how the programmer can interact with StarPU to easily express (basic) algorithmic knowledge to help the scheduling engine. It should be noted that StarPU is able to exploit such hints in a portable way since we observe similar behavior with different platforms, even though the potential load imbalance heavily depends on the relative speedups of the different processing units too.

4.4. Taking advantage of heterogeneity

In our *heterogeneous* context, we define the **efficiency** as the ratio between the sum of the computation powers obtained separately on each architecture and the computation power obtained while using all architectures at the same time. This indeed expresses how well we manage to add up the powers of the different architectures. In the case of homogeneous processors, the total power should usually not exceed the sum of the powers obtained by the different processing units. In the heterogeneous case there can however be some computation affinities: a GPU may be perfectly suited for some type of task while another type of task will hardly be as efficient as on a CPU. The second column of Figure 4(b), for instance, shows that the different tasks that compose an LU decomposition do not perform equally: matrix products (DGEMM) are especially efficients while the pivoting is quite inefficient with a mere speedup of 1.2. It is worth noting that the relative speedups are rather limited because the CPU double precision BLAS implementation (ATLAS) is heavily optimized and nearly reaches peak performance, whereas the CUBLAS library (and QUADRO FX5800 hardware) are still not fully optimized for double precision.

Figure 4(a) shows that in some case, the speed obtained by a hybrid platform outperforms the sum of the speeds of the elements that compose the system: this is what we denote as a *superlinear efficiency*. Various factors must be considered to understand how StarPU achieves this result. First, idleness, which could be a major cause of inefficiency, is heavily reduced by the use of performance models that permit very good load balancing. The overhead directly introduced by StarPU to execute a task (typically a couple of *s*) is also small compared with the overhead of CUDA itself: launching a kernel or initiating a memory transfer usually takes at least half a dozen *s* [7]. Not only StarPU executes tasks rapidly while avoiding load imbalance, but the use of per-task performance models allows the **heft-tm** strategy to distribute tasks with respect to their actual efficiency on the different processing units. Figure 4(c) shows that tasks tend to

	(3 CPUs + 1 GPU)	3 CPUs	1 GPU
GFlop/s	79.5 ± 0.4	17.4 ± 0.3	61.0 ± 0.1
Efficiency	$79.5 = \mathbf{101.3\%}$ (17.4	$+$	61.0)

(a) Performance of DGETRF on a 3.2 GB problem.

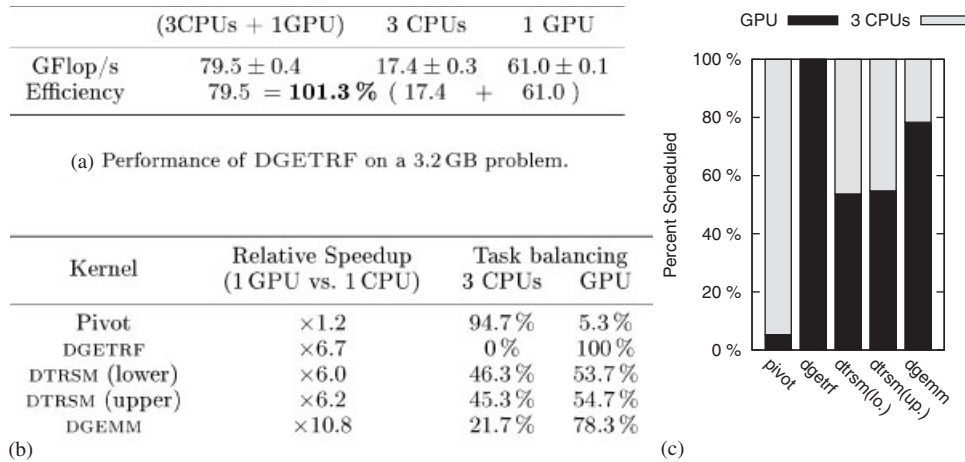| Kernel	Relative Speedup (1 GPU vs. 1 CPU)	Task balancing 3 CPUs	GPU
Pivot	$\times 1.2$	94.7%	5.3%
DGETRF	$\times 6.7$	0%	100%
DTRSM (lower)	$\times 6.0$	46.3%	53.7%
DTRSM (upper)	$\times 6.2$	45.3%	54.7%
DGEMM	$\times 10.8$	21.7%	78.3%

(b)

(c)

Figure 4. Example of a superlinear acceleration on a double precision LU decomposition with partial pivoting (DGETRF) applied on a 3.2 GB matrix: (a) performance of DGETRF on a 3.2 GB problem; (b) relation between speedup and task distribution; and (c) task balancing.
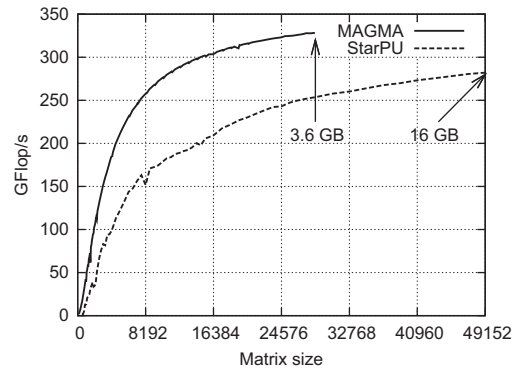
Figure 5. SGETRF with MAGMA and StarPU.

be executed where they are the most efficient: matrix products are more likely to be offloaded on the GPU while the resolution of triangular system is relatively more efficient on CPUs. It is worth noting that the diagonal block factorization is always performed on the GPU because these tasks, which take a very long time, are in the critical path of the parallel algorithm. To sum up, StarPU reduces the impact of *Amdhal's Law* with a limited parallelization overhead, and the use of per-task performance models allows the scheduling engine to dispatch the tasks where they are the most efficient while preserving parallelism. That is why it is possible to observe a superlinear efficiency which means *StarPU transparently takes advantage of the heterogeneous nature of the platform without any support from the programmer*.

4.5. Comparing with a reference hand-tuned library

Figure 5 shows the speed of an LU decomposition with partial pivoting (SGETRF) running on a single nehalem CPU aside with a QUADRO FX5800 GPU, either with StarPU or with the MAGMA library [8]. Contrary to our implementation, MAGMA uses a static scheduling and a heavily hand-tuned implementation of the SGETRF algorithm. Both implementations use the kernels from the CUBLAS library on the GPU side, and rely on GotoBLAS for the CPU side. Our kernels however call them in a straightforward non-optimized way. Unlike StarPU, MAGMA is specifically optimized for small problems. In the case of medium-sized problems, MAGMA is typically *only* 25% faster than our simple (and portable) implementation of the algorithm. However, MAGMA requires that the entire matrix fits into the memory of the GPU, to make it possible to

efficiently solve the problem in a static fashion. On the other hand, StarPU transparently factorizes arbitrary large matrices thanks to our dynamic scheduling approach. Given the considerable efforts required to develop the MAGMA library, this illustrates how our dynamic approach enables portable performance along with a significant improvement in terms of productivity.

5. RELATED WORK

Recent years have witnessed the democratization of accelerator-based computation. Most research efforts have been devoted to generating and offloading efficient code for accelerators, and only a few projects have investigated the problem of tasks scheduling over accelerator-based machines though. To our knowledge, our work is the first to provide an in-depth study of portable task scheduling on such heterogeneous machines equipped with accelerators.

Charm++ is a parallel C++ library that provides sophisticated load balancing and communication optimization mechanisms. Its runtime system implementation supports GPUs [9] and Cell processors [10]. Even though the Offload API provided by Charm++ is adopted by most existing task-based approaches, there is currently no performance evaluation available yet, to the best of our knowledge.

In [11], Diamos *et al.* present a collection of techniques for the **Harmony** runtime system which they plan to implement in a complete system. Some of those techniques are similar to those implemented in StarPU. For instance, they consider the problem of task scheduling with the support of performance models. However, Harmony does not accept user-provided scheduling strategies. Besides the regression-based model also proposed by Harmony, the tight integration of StarPU's data management library along with the scheduler allows much simpler, yet more accurate, history-based performance models [2]. The data management facilities offered by StarPU are also much more flexible as it is possible to manipulate data in a high-level fashion that is much more expressive than a mere list of addresses.

StarSs introduces `#pragma` language annotations similar to those proposed by Ayguadé *et al.* [12]. They rely on a source-to-source compiler to generate offloadable tasks. Contrary to StarPU, the implementation of the StarSs model is done by the means of a separate runtime system for each platform (SMPSs, GPUSs [13], CellSs [14]). Some efforts are done to combine those different runtime systems: Planas *et al.* allow programmers to include CellSs tasks within SMPSs tasks [15], but this remains the duty of the programmer to decide which tasks should be offloaded. In contrast, StarPU considers tasks that can indifferently be executed on multiple targets.

The approach of the **Anthill** runtime system is very similar to StarPU. Teodoro *et al.* experimented [16] the impact of task scheduling for clusters of machine equipped with a single GPU and multiple processors. They implemented two scheduling policies which are equivalent to the simple greedy strategies we have shown in Table I. The tight integration of data management and task scheduling within StarPU makes it possible to tackle the problem of task scheduling onto accelerators while taking data movements overhead into account, while Anthill only considers relative speedups to select the most appropriate processing unit.

6. CONCLUSION AND FUTURE WORK

We presented StarPU, a new runtime system that efficiently exploits heterogeneous multicore architectures. StarPU and all the experiments presented in this paper can be downloaded at http://runtime.bordeaux.inria.fr/StarPU/. It provides a uniform execution model, a high-level framework to design scheduling policies, and a library that automates data transfers. We have written several scheduling strategies and observed how they perform on some classical problems.

In addition to improving programmability by the means of a high-level uniform approach, we have shown that applying simple scheduling strategies can significantly reduce load balancing issues and improve data locality. Since there exists no ultimate scheduling strategy that addresses all application needs, or even address all inputs for a given application, StarPU permits to dynamically

select the strategies at runtime, thus letting the programmer, e.g. try and choose the most efficient strategy. This makes it possible to benefit from scheduling without putting restrictions or making excessive assumptions. We also demonstrated that given a proper scheduling, it is possible to exploit the specificity of the various computation units of a heterogeneous platform and to obtain a consistent superlinear efficiency.

It is crucial to offer such uniform abstraction of the numerous programming interfaces that result from the advent of accelerator technologies. Unless such a common approach is adopted, it is very unlikely that accelerators will evolve from a *niche* with dispersed efforts to an actual mainstream technique. While the OPENCL standard also does provide task queues and an API to offload tasks, our work shows that such a programming model needs to offer an interface that is simple but also expressive.

We plan to implement our model on other accelerator technologies, for instance, by the means of an OPENCL driver. StarPU should also support platform equipped with multiple GPUs. In the future, we expect compiling environments, such as HMPP [17], to generate efficient tasks. We are also planning to port real applications such as the MAGMA library. Last but not least, StarPU could reduce the gap between HPC applications and theoretical works in the field of scheduling by providing a high-level platform to implement advanced scheduling policies.

REFERENCES

1. Augonnet C, Namyst R. A unified runtime system for heterogeneous multicore architectures. *Proceedings of the International Euro-Par Workshops*, *HPPC'08*, Las Palmas de Gran Canaria, Spain, August 2008.
2. Augonnet C, Thibault S, Namyst R, Nijhuis M. Exploiting the Cell/BE architecture with the StarPU unified runtime system. *SAMOS Workshop*, Samos, Greece, July 2009.
3. Topcuoglu H, Hariri S, Wu M-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 2002; **13**(3):260–274.
4. Banino C, Beaumont O, Carter L, Ferrante J, Legrand A, Robert Y. Scheduling strategies for master–lave tasking on heterogeneous processor platforms. *IEEE Transactions on Parallel and Distributed Systems* 2004; **15**(4):319–330.
5. Clint Whaley R, Dongarra J. Automatically tuned linear algebra software. *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Jose, CA, U.S.A., 1999.
6. Augonnet C, Thibault S, Namyst R. Automatic calibration of performance models on heterogeneous multicore architectures. *Proceedings of the International Euro-Par Workshops*, *HPPC'09*, Delft, The Netherlands, August 2009.
7. Volkov V, Demmel JW. Benchmarking GPUs to tune dense linear algebra. *SC '08 Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press: Piscataway, NJ, U.S.A., 2008; 1–11.
8. Tomov S, Dongarra J, Baboulin M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Technical Report*, January 2009.
9. Wesolowski L. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. *Master's Thesis*, 2008.
10. Kunzman D, Zheng G, Bohm E, Kalé LV. Charm++, offload API, and the cell processor. *Proceedings of the PMUP Workshop*, Seattle, WA, U.S.A., September 2006.
11. Diamos GF, Yalamanchili S. Harmony: An execution model and runtime for heterogeneous many core systems. *HPDC '08 Proceedings of the 17th International Symposium on High Performance Distributed Computing*. ACM: New York, NY, U.S.A., 2008; 197–200.
12. Ayguade E, Badia RM, Cabrera D, Duran A, Gonzalez M, Igual F, Jimenez D, Labarta J, Martorell X, Mayo R, Perez JM, Quintana-Ortí ES. A proposal to extend the openmp tasking model for heterogeneous architectures. *IWOMP '09*: *Proceedings of the 5th International Workshop on OpenMP*, Berlin, Heidelberg, 2009, 154–167.
13. 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*, Delft, The Netherlands, August 2009.
14. Bellens P, Pérez JM, Badia RM, Labarta J. Exploiting locality on the Cell/B.E. through bypassing. *SAMOS*, Samos, Greece, 2009; 318–328.

15. Planas J, Badia RM, Ayguadé E, Labarta J. Hierchical task based programming with StarSs. *International Journal of High Performance Computing Application* 2009; **23**:284.
16. Teodoro G, Sachetto R, Sertel O, Gurcan M, Meira Jr W, Catalyurek U, Ferreira R. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. *Proceedings of the IEEE International Conference on Cluster Computing*, New Orleans, LA, U.S.A., 2009.
17. Dolbeau R, Bihan S, Bodin F. HMPP A hybrid multi-core parallel programming environment. *Technical Report*, CAPS entreprise, 2007.