StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines

Cédric Augonnet, Samuel Thibault, Raymond Namyst

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines

Cédric Augonnet, Samuel Thibault, Raymond Namyst

N° 7240

Mars 2010

ISSN 0249-6399 ISRN INRIA/RR--7240--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
BORDEAUX - SUD OUEST

StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines

Cédric Augonnet*, Samuel Thibault†, Raymond Namyst‡

Theme :
Équipe-Projet Runtime

Rapport de recherche n° 7240 — Mars 2010 — 30 pages

Abstract: Multicore machines equipped with accelerators are becoming increasingly popular. The TOP500-leading RoadRunner machine is probably the most famous example of a parallel computer mixing IBM Cell Broadband Engines and AMD opteron processors. Other architectures, featuring GPU accelerators, are expected to appear in the near future. To fully tap into the potential of these hybrid machines, pure offloading approaches, in which the main core of the application runs on regular processors and offloads specific parts on accelerators, are not sufficient. The real challenge is to build systems where the application would permanently spread across the entire machine, that is, where parallel tasks would be dynamically scheduled over the full set of available processing units.

To face this challenge, we propose a new runtime system capable of scheduling tasks over heterogeneous, accelerator-based machines. Our system features a software virtual shared memory that provides a weak consistency model. The system keeps track of data copies within accelerator embedded-memories and features a data-prefetching engine. Such facilities, together with a database of self-tuned per-task performance models, can be used to greatly improve the quality of scheduling policies in this context. We demonstrate the relevance of our approach by benchmarking various parallel numerical kernel implementations over our runtime system. We obtain significant speedups and a very high efficiency on various typical workloads over multicore machines equipped with multiple accelerators.

Key-words: GPGPU, Multicore, Scheduling, Performance, StarPU.

* Cédric Augonnet is with the INRIA and the University of Bordeaux, E-mail: cedric.augonnet@inria.fr

† Samuel Thibault is with the INRIA and the University of Bordeaux, E-mail: samuel.thibault@labri.fr

‡ Raymond Namyst is with the INRIA and the University of Bordeaux, E-mail: raymond.namyst@labri.fr

Centre de recherche INRIA Bordeaux – Sud Ouest
Domaine Universitaire - 351, cours de la Libération 33405 Talence Cedex
Téléphone : +33 5 40 00 69 00

StarPU: un support exécutif capable d'ordonnancer des tâches sur des machines multicœurs équippées d'accélérateurs

Résumé : Les machines multicœurs équippées d'accélérateurs sont de plus en plus répandues. Le RoadRunner, une des machines parallèles actuellement les plus puissantes au monde est d'ailleurs constitué d'une combinaison de processeurs IBM Cell Broadband Engines et d'AMD Opteron. D'autres architectures à base d'accélérateurs (notamment de GPUs) devraient également faire leur apparition dans un futur proche. Afin de réellement tirer la quintessence de ces machines hybrides, il n'est plus suffisant de se contenter d'approches uniquement fondées sur le déport de calcul, où le cœur de l'application tourne sur des processeurs classiques alors que les parties véritablement coûteuses sont déportées sur des accélérateurs. Le véritable défi consiste à concevoir des systèmes où les applications sont réparties sur la totalité de la machine, c'est à dire des qu'il faut être capable d'ordonnancer des tâches parallèles sur la totalité des ressources de calcul disponibles.

Afin de faire face à ce défi, nous présentons un nouveau support exécutif capable d'ordonnancer des tâches sur des machines hétérogènes à base d'accélérateurs. Notre système comporte une mémoire virtuellement partagée qui assure la cohérence des données. Il garde la trace des copies de chacune des données dans les différentes mémoires embarquées sur les accélérateurs, et fourni des mécanismes tels que le pré-chargement de données. Conjointement à l'utilisation de modèles de coût auto-calibrés, ces fonctionnalités permettent d'améliorer substantiellement la qualité de l'ordonancement dans le contexte des architectures hybrides. Nous démontrons l'intérêt de notre approche en analysant les performances de notre support exécutif sur différents algorithmes numériques parallèles. Nous observons une réduction significative des temps de calcul, ainsi qu'une grande efficacité dans l'utilisation des différentes ressources de calcul pour différentes charges de travail typiques, notamment dans le cas de machines multicœurs équippées de plusieurs accélérateurs.

Mots-clés : GPGPU, Multicœur, Ordonnancement, Performance, StarPU.

1 Introduction

The High Performance Computing community has recently witnessed a major evolution of parallel architectures. The invasion of multicore chips has impacted almost all computer architectures, going from laptops to high-end parallel computers. While researchers are still having a hard time trying to bridge the gap between the theoretical performance of multicore machines and the sustained performance achieved by current software, they now have to face yet another architecture trend: the use of specific purpose processing units as side accelerators to speed up computation intensive applications.

The Cell Broadband Engine processor, for instance, features a general purpose core (PowerPC Processing Unit) surrounded by eight co-processors (Synergistic Processing Units), each featuring a specific vector instruction set and equipped with a private local storage. This processor is currently used in several multimedia hardware solutions – it most notably powers the Sony PlayStation 3 – but it is also a key component of the currently most powerful parallel machine: IBM RoadRunner. Although the future of the Cell processor in HPC is not clear, many computer architects predict that future processors will follow a similar design mixing general purpose cores and specialized co-processors.

Another popular practice is to use General Purpose Graphical Processing Units as co-processors to speed up data parallel computations. GPUs are massively parallel devices that can run SIMD programs very efficiently. Many research groups have already reported impressive performance boost thanks to the offloading of part of their application to a GPU device. Although the generalization to multi-GPU computation is currently limited by the bandwidth of the I/O buses connecting these devices to the host machine, some computer manufacturers actually plan to build HPC architectures featuring multiple GPUs racks. It is however expected that GPU-like co-processors and general purpose cores will be part of the same chip in the near future. The future Intel Larrabee processor, for instance, clearly belongs to that category of hybrid processors.

As a result, for an increasing part of the HPC community, the challenge has somehow moved from exploiting hierarchical multicore machines to exploiting heterogeneous multicore architectures. In the former case, many research efforts are merely dedicated to enhancing existing compilers, runtime systems or parallel libraries to better meet the requirements of new multicore architectures. Regarding heterogeneous machines, the gap with traditional multiprocessor architectures is so huge that simply enhancing existing solutions is not an option. Accelerator-based machines indeed introduce many hardware constraints – related to several memory limitations and to the execution model of accelerators – that make traditional multithreaded approaches impractical. Indeed, accelerators such as GPUs or Cell's SPUs have their own local storage which is not implicitly kept consistent with the main host memory. Thus, all data transfers between the main memory and the accelerators must be explicitly done by the software. Furthermore, because accelerators feature their own instruction set and a specific execution model, the code offloaded to these co-processors must be generated by a specific compiler and is completely different from the code compiled for the regular cores, which makes it impossible to migrate tasks between processors and accelerators during their execution.

As a result, exploiting this new generation of parallel computers requires to revise the complete software stack, from the languages down to the runtime

systems. Our contribution is the design of a new runtime system, called StarPU, that meets the needs of heterogeneous, accelerator-based architectures. We explain how StarPU is capable of scheduling tasks efficiently over the whole range of processing units, thanks to the use of auto-tuned performance models and optimized data transfers between processing units. Our runtime system is typically intended to serve as a target for numerical kernel libraries and parallel language compilers. We use the StarPU implementation over NVIDIA GPU cards and Intel Nehalem processors to perform detailed performance studies of our scheduling techniques on several numerical kernels. The results of these studies provide insights into the impact of the accuracy of performance models and the granularity of the data partitioning.

2 Background

We first review the main types of approaches to the problem of programming accelerator-based machines.

2.1 New programming tools

To deal with accelerator programming, a wide spectrum of languages, techniques and tools that have been specifically designed, ranging from low-level development kits to new parallel languages.

2.1.1 Low-level development kits

Constructors usually provide low-level software development toolkits (SDK) to program their particular type of accelerators. Since such accelerator-specific APIs are usually the first available for the corresponding acceleration technologies, they are usually also the most widely used. CUDA, for instance, is the official SDK provided by NVIDIA for their GPU cards, while the AMD company provides Stream, a totally different SDK for their ATI cards. Concerning programming the Cell, LibSPE provided by IBM still remains the main SDK.

All these SDKs look similar at first sight, but they also export many hardware capabilities to the application, allowing to develop very efficient, carefully tuned code. However, their low-level API is hard to use and leads to non-portable programs. They also do not provide support for distributing work between regular cores and accelerators, they merely provide tools to drive computations on the latter.

The recently announced OpenCL standard provides a portable low-level interface, but it does not really help people in handling the complexity of such challenging hardware: such kind of low-level library does not really take any decision, and just obeys to the programmer's orders. In order to achieve the best performance a programmer would still have to know the intricacies of the underlying hardware, to give judicious orders.

2.1.2 Optimized Kernel Libraries

Because many HPC applications are making extensive use of common numerical kernels (BLAS for linear algebra [24], FFT for Fourier transforms [13]), it is no surprise that numerous research groups have designed implementations

of numerical kernel libraries specific to GPU or Cell accelerators [19]. These highly optimized libraries are typically built using the aforementioned software development toolkits. For instance, the CUBLAS library provides optimized BLAS routines for NVIDIA GPU cards, and is implemented in CUDA.

Most existing libraries are only capable of exploiting a single accelerator at a time, though. Some libraries such as MAGMA consider not only multi-accelerator setups, but they are also currently getting support for using CPUs and accelerators simultaneously [22]. However, their implementation is based on a static distribution of tasks and data, which would not be suitable for complex systems composed of multiple heterogeneous accelerators. Furthermore, because existing implementation do not feature a true task scheduler, a parallel numerical kernel can not be run in parallel with another instance of a kernel without resulting to interferences. This problem prevents efficient parallel code composability.

2.1.3 Programming Languages

Several new specific languages have been designed to write portable programs in a high-level fashion that should take advantage of accelerator-based machines.

Many new languages are based on a streaming model. RapidMind and BrookGPU provide convenient ways to explicitely manipulate data streams. Other languages such as Sequoia or the MapReduce framework let the programmer express computations following some structure that exposes a sufficient amount of parallelism.

There are also numerous propositions to extend existing languages with some OpenMP-like constructs. StarSs and CellGen both propose a source-to-source approach to perform target-specific code optimization (eg. vectorization or tiling) and propose pragmas to delimit tasks and describe data accesses. Similarly, PGI ACCELERATOR and HMPP also take care of transforming annotated C/Fortran codes into efficient kernels for accelerators.

While those efforts attempt to save programmers from low-level architecture-specific interfaces thanks to high-level and portable constructs, most of them concentrate on generating efficient code. For instance, HMPP [10] relies on a minimalist runtime system that simply uses a coarse-grain lock to protect each processing units. Some languages such as Sequoia even map computations in a static fashion [11]; this may not be realistic when it comes to irregular applications and/or complex heterogeneous platforms. Runtime systems permit to do dynamically what cannot be done statically anymore.

2.2 Towards new runtime systems

The numerous research efforts presented in the previous sections cover a wide range of functionalities, but still suffer from relying on basic runtime systems. We claim that implementations of numerical kernel libraries or parallel languages could achieve better performance over complex heterogeneous architectures if they would use a runtime system *capable of dynamically scheduling tasks on the most appropriate processing unit.*

Designing such a runtime systems raises a number of challenges related both to the scheduling itself and to the management of data movements between the host memory and the accelerators.

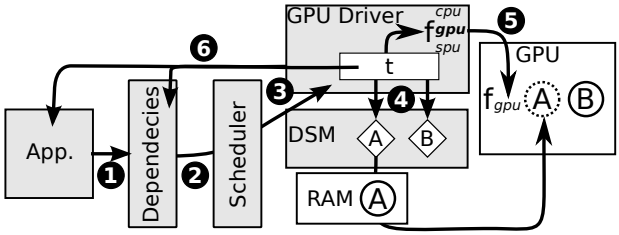


Figure 1: Execution of a Task within StarPU.

Scheduling. To allow the scheduler to dynamically assign tasks to any processing unit, tasks should have several implementations, one per type of processing unit that is capable of executing it. These implementations will obviously exhibit very different performance, but the gap between the implementation for an Intel Nehalem processor and the implementation for an NVIDIA Quadro GPU can also vary a lot depending on the kind of computation itself. Thus, the scheduler should be able to use performance prediction information as an input for its scheduling policy, so as to compute the best possible task assignment. Moreover, we think that the ability to easily tune and optimize scheduling policies incrementally is fundamental on this new generation of architecture, as we will show in the following sections.

Data Management. Data management was already a delicate issue on multicore machines, due to the *Non-Uniform Memory Accesses* introduced by the topology formed by the memory banks. On archictectures equipped with accelerators, the situation is even worse, especially if the accelerators are connected via a slow I/O bus. Numerous data movements between the main memory and the embedded memory of an accelerator can really hurt performance. Thus, it is crucial that the runtime system keeps track of data copies linked to each processing unit so as to avoid unnecessary data movement whenever possible. Moreover, the runtime can go much further and estimate the data transfer times to better compare different scheduling alternatives, or even provide a powerful data management system able to optimize transfers of complex data structures (cyclic column distributions, etc.) and perform data prefetching to overlap part of the transfer cost.

The next section presents our answers to these challenges.

3 The StarPU runtime system

StarPU is a runtime system for scheduling a graph of tasks onto a heterogeneous set of processing units, intended to provide the powerful features detailed in Section 2.2 in a portable but still efficient way. It is meant to be used as a backend for *e.g.* parallel language compilation environments and High-Performance libraries, here called *the application*. The two basic principles of StarPU is firstly that tasks can have several implementations, for some or each of the various heterogeneous processing units available in the machine, and secondly that necessary data piece transfers to these processing units are handled transparently by StarPU.

Figure 1 gives an overview of the journey of tasks within StarPU, thus providing a general idea of how StarPU modules are related. The application first submits tasks (1). When a task gets ready (2), it is dispatched to one of the device drivers by the scheduler (3). The DSM ensures the availability of all pieces of data (4). The driver then offloads the proper implementation for the task (5). When the task completes, tasks depending on it are released and an application callback for the task is executed (6). Before further sections provide more details, the following subsections give the rationale for the DSM and the scheduler modules, as well as a complete StarPU programming example.

3.1 A virtually shared memory system for heterogeneous chips

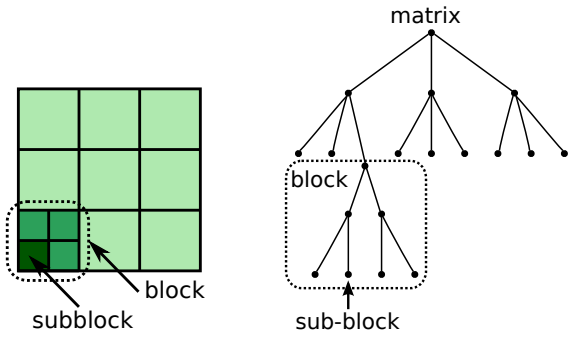
Because StarPU schedules tasks at runtime, data transfers have to be done automatically and "just-in-time" between processing units, relieving the application programmer from explicit data transfers. Moreover, to avoid unnecessary transfers, StarPU keeps data where it was last needed, even if was modified there, and it allows multiple copies of the same data to reside at the same time on several processing units as long as it is not modified. To summarize, the data management part of StarPU, detailed in Section 4, actually implements a Software DSM (Distributed Shared Memory), with relaxed consistency and data replication capability thanks to an MSI (Modified/Shared/Invalid) protocol. The application just has to register the different pieces of data by giving their addresses and sizes in the main memory. Data can thus be dynamically as well as statically allocated.

StarPU also provides an additional high-level abstraction to easily handle partitioned data for block- and tile-based parallelism: *filters*. These can be used to logically divide data into smaller blocks, according to the application algorithm needs (*e.g.* MAGMA [22]). For instance, Figure 2 illustrates how a matrix can be first divided into $3 * 3$ blocks, and then one of these blocks can be further divided into $2 * 2$ sub-blocks, resulting in a tree of sub-data. Multiple filters are provided and can be extended to match various splitting strategies (bloc, cyclic, etc.) and various native memory access patterns (dense BLAS matrices, sparse CSR matrices, etc.). Such filters can be applied dynamically at runtime, to stick to the algorithm behavior, in particular application featuring adaptive refinement methods, and allow the DSM to transfer to the processing unit only the exact needed data, or to find it inside an already-transferred data containing it.

3.2 A tasking model enabling heterogeneous scheduling

StarPU tasks can be executed by as many processing units as possible, those which the programmer has provided an implementation for, including accelerators which do not have access to the main memory. Therefore, the programmer has to also explicit all the input and output data of tasks by using the handles returned by the DSM, so that the latter can automatically fetch data as needed right before execution.

The submission of tasks is asynchronous and termination is signaled through a callback. This lets the application submit several tasks, including tasks which depends on others. To express dependencies and thus actual graphs of tasks,

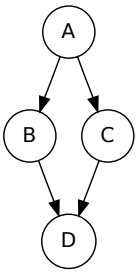


(a) A partitioned piece of data.

(b) Its tree representation.

```
/* register the matrix to StarPU */
h = register_matrix(&matrix, ptr, n, n, ...);

/* divide the matrix into 3x3 blocks */
map_filters(matrix, 2, filter_row, 3, filter_col, 3);

/* divide the bottom lower block (2,0)
 * into 2x2 subblocks */
block = get_sub_data(mat, 2, 2, 0);
map_filters(block, 2, filter_row, 2, filter_col, 2);

/* bottom left sub-block (1,0) */
subblock = get_sub_data(block, 2, 1, 0);
```

Figure 2: An example of partitioned data, its tree representation and the corresponding StarPU code




```
declare_deps(tagB, 1, tagA);
declare_deps(tagC, 1, tagA);
declare_deps(tagD, 2, tagB, tagC);

taskA->tag_id = tagA;
(...)
taskD->tag_id = tagD;

submit_task(taskA);
(...)
submit_task(taskD);

tag_wait(tagD);
```

Figure 3: Expressing task dependencies with tags

```
1  void cublas_SGEMM(data_interface_t *descr, void *arg) {
2    float *subA = descr[0].matrix.ptr;
3    float *subB = descr[1].matrix.ptr;
4    int nxA = descr[0].matrix.nx;
5    (...)
6    cublasSgemm(... subA, nxA, subB, ...);
7  }
8
9  void cpu_SGEMM(data_interface_t *descr, void *arg) {
10   float *subA = descr[0].matrix.ptr;
11   float *subB = descr[1].matrix.ptr;
12   int nxA = descr[0].matrix.nx;
13   (...)
14   SGEMM(... subA, nxA, subB, ...);
15 }
16
17 static starpu_codelet SGEMM_cl = {
18   .where = CORE|CUDA,
19   .cpu_func = cpu_SGEMM,
20   .cuda_func = cublas_SGEMM,
21   .nbuffers = 3
22 };
```

Figure 4: An example of codelet implementing a matrix product (GEMM).

integers called *tags* are used to characterize the termination of some task. Figure 3 illustrates how the dependencies between 4 tasks can be expressed. The advantage of decoupling tags from the tasks themselves is that a tag can depend on another tag which has not been associated with a task yet, or on another tag whose task is already finished. This lets the application avoid having to submit all the tasks at once (because it would completely fill the memory or just does not make sense, in the case of flow applications). A tag can also be explicitly waited for by the main application to synchronize with the execution of StarPU tasks.

3.3 Developing on top of StarPU: a case study

Figure 6 gives a global idea of how using StarPU looks like. It shows the implementation of a blocked matrix multiplication taking advantage of all CPUs and GPUs by leveraging both the existing CUBLAS and BLAS SGEMM kernels at the same time.

On the left part, the CUBLAS and BLAS SGEMM operations are first wrapped into generic functions to harmonize the calling convention, and the `cl` structure groups them into just one *codelet* which will take 3 matrix parameters and which is able to run on either a CPU or a GPU.

The right part shows how to submit tasks executing that codelet to perform the computation block per block. The `A`, `B`, and `C` matrices are first registered, then partitioned into blocks as represented on the bottom left figure: `N` rows and `M` columns. A 2-dimension loop then creates `N*M` tasks executing the SGEMM codelet and assigns to them the proper sub-parts of the matrices. After waiting

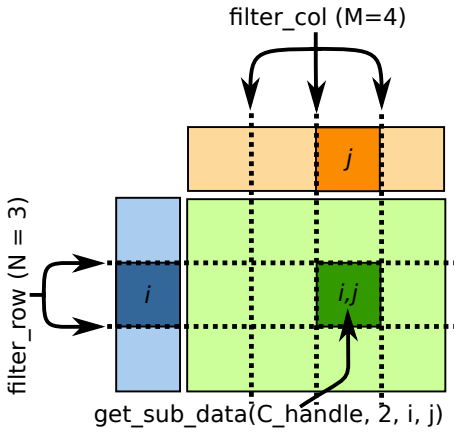


Figure 5: Manipulating sub-matrices with filters.

for the termination of all tasks, the `C` matrix can be unpartitioned and synchronized, to make sure parts computed on GPUs are written back to the main memory.

It is worth noting that this `fast_SGEMM` function, instead of waiting for task completion itself, could also just return a tag depending on the termination of all tasks, thus allowing the application to start other kinds of computations concurrently, and StarPU to schedule them as efficiently as possible along the SGEMM kernels. Also, such kind of code could be automatically generated for a range of BLAS operations.

4 Managing Data

We now detail the internal principles of the data management module of StarPU, in particular the DSM mechanisms being used.

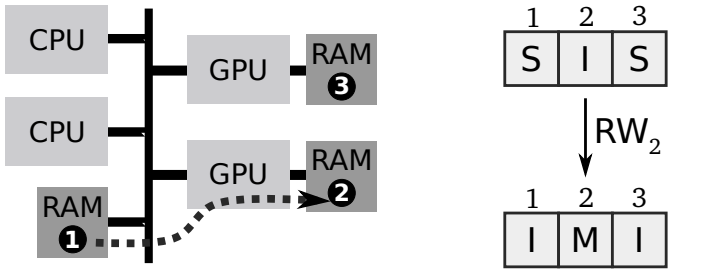
4.1 Data replication and memory consistency

When the application registers some data to the DSM, StarPU allocates in the main memory an array to record the MSI states of that data on the different memory nodes, as illustrated by Figure 7(b). This permits to replicate the data on multiple memory nodes (multiple-reader Shared access) or to keep a modified version on one memory node (single writer Modified access) while keeping read-write consistency thanks to the MSI protocol. Here, for instance, the memory node number 2, a GPU, requests both Read and Write access, and it does not have a copy of the data (Invalid state). The data is transferred for the Read access, and other copies are invalidated due to the requested Write access.

By default, modified data is not immediately written back to the main memory. This lazy approach permits StarPU to save bus bandwidth when data is re-used by several tasks. If however the programmer knows that the data will not be re-used he can request for a write-through strategy. When enough memory is not available on some processing unit for the data required by a task,

```
void fast_SGEMM(float *A, float *B, float *C, int N,
    int M) {
  /* Register the matrices */
  register_matrix(&A_handle, A, ...);
  register_matrix(&B_handle, B, ...);
  register_matrix(&C_handle, C, ...);

  /* Partition data  */
  partition_data(A_handle, filter_row, N);
  partition_data(B_handle, filter_col, M);
  map_filters(C_handle, 2, filter_row, N, filter_col, M
      );

  /* Submit tasks */
  for (i = 0; i < N; i++) //loop on cols
    for (j = 0; j < M; j++) //loop on rows
    {
      task = starpu_task_create();
      task->cl = &SGEMM_cl;

      task->buffers[0].handle =
        get_sub_data(A_handle, 1, j);
      task->buffers[0].mode = STARPU_R;
      task->buffers[1].handle =
        get_sub_data(B_handle, 1, i);
      task->buffers[1].mode = STARPU_R;
      task->buffers[2].handle =
        get_sub_data(C_handle, 2, i, j);
      task->buffers[2].mode = STARPU_RW;

      submit_task(task);
    }

  wait_all_tasks();

  // make matrix C available to the app.
  unpartition_data(C_handle);
  sync_data_with_mem(C_handle);
}
```

Figure 6: Blocked matrix multiplication written with StarPU using the codelet defined in Figure 4 and the filters described on Figure 5.



(a) Data transfer to a GPU. (b) Data state is updated.

Figure 7: The MSI protocol maintains the state of each data on the different memory node. This state (Modified, Shared or Invalid) is updated accordingly to the access mode (Read or Write).

StarPU performs a memory reclaiming pass to flush some of the useless data out.

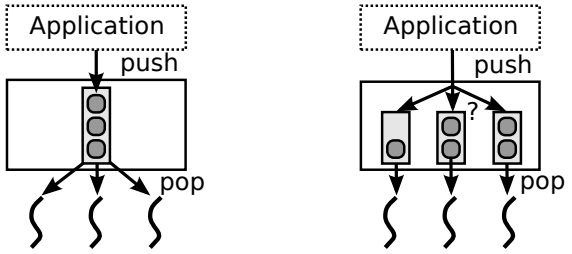
As a result, even for a complex application with a graph of tasks dealing with various data whose total size can not fit into the local memory of processing units and for which having to issue data transfers by hand would be very difficult, StarPU lets the programmer just express the blocked or tiled computation tasks and data dependencies, and StarPU will transparently optimize the usage of the limited local memory.

4.2 Asynchronous data management and data requests

Data transfers can be very long due to the main bus typically being a bottleneck. Nowadays acceleration cards support asynchronous data transfer, *i.e.* these transfers can be overlapped with computations. StarPU's data management is also completely asynchronous by associating each memory node with a queue of pending data requests, possibly queued by various parts of StarPU, another memory node requesting data for some task for instance. It also makes dynamic prefetching of data quite natural: as soon as a task is scheduled to be run by some processing unit, the data transfer order can be queued so that the execution of the task can hopefully happen as soon as the processing unit has finished previous tasks, thanks to data being transferred in parallel. More generally the scheduling and the data management library of StarPU can collaborate to optimize the whole execution.

5 Scheduling tasks and data transfers

Because a generic scheduler can not achieve the best performance for all kinds of application with all hardware, StarPU aims at hosting a testbed to implement and choose between various powerful scheduling strategies in a portable way. This section details how schedulers can be implemented and describes a few scheduler examples already provided along StarPU.



(a) A portable interface. (b) An example of strategy.

Figure 8: All scheduling strategies implement the same queue-based interface.

Table 1: Scheduling strategies implemented with StarPU

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	HEFT based on Task duration Models
heft-tm-pr	**heft-tm** with data PRefetch
heft-tmdp	**heft-tm** with remote Data Penalty
heft-tmdp-pr	**heft-tmdp** with data PRefetch

5.1 A portable scheduling framework

The role of the StarPU scheduler is to dispatch tasks onto the different processing units. From the point of view of the application, tasks are just *pushed* to the scheduler, while from the point of view of the drivers of the processing units, tasks are just *poped* from the scheduler, i.e. the simplest implementation is a single FIFO queue as is shown on Figure 8(a). A more complex scheduler, like the one illustrated in Figure 8(b) thus can boil down to organizing a set of queues (which can be FIFOs, stacks, deques, etc.) and implement the above-mentioned *push* and *pop* operations. All distribution, load balancing, etc. decisions are matters internal to the scheduler. This approach permits to design various schedulers independently of the application and the drivers being used, and to choose which one should be used at runtime. Table 1 gives a list of the different portable policies that have been implemented in StarPU and which will be described in the following subsections. More details can be found in a previous paper [3].

5.2 Greedy Strategies

The simplest strategy consists in having all processing units share a single FIFO as illustrated in Figure 8(a). A refinement of this is to make the push method of the FIFO take programmer-provided priorities of tasks into account: more prioritized tasks will be queue closer to the output of the FIFO, resulting to the **greedy** StarPU scheduler, as opposed to the **no-prio** scheduler.

Such strategy however offers poor scalability w.r.t. the number of accelerators due to locking contention, especially in case the tasks are fine grained. This

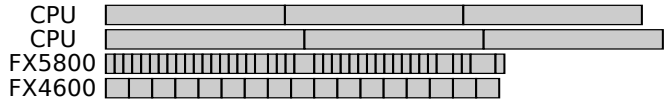


Figure 9: A pathological case: Gantt diagram of a blocked matrix multiplication with a greedy strategy.

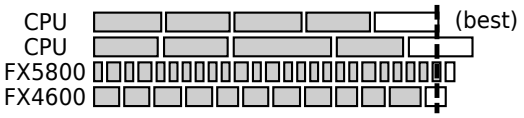


Figure 10: The Heterogeneous Earliest Finish Time Strategy.

is commonly solved by decentralizing task queues as shown in Figure 8(b). A Cilk-like [12] work-stealing strategy was implemented in the pop method of the scheduler to steal tasks from other queues if the local queue is empty, resulting to the **ws** scheduler.

These schedulers are very portable in that they work with generic tasks. However, as discussed in details later in Section 6, the completion time of tasks can very often be predicted by automatic cost models, at least with rough precision. Greedy strategies can get huge benefit from this kind of information.

5.3 Cost model-guided scheduling strategies

Figure 9 shows an typical example of the distribution of tasks by the **greedy** or **ws** scheduler onto a dualcore machine equipped with two GPUs with differing performance. The heterogeneity of performance between different processing units leads to a load imbalance, which could have been avoided if the scheduler knew that task completion takes so much time on CPUs.

A common approach to tackle it is to consider the computational power of each unit (*e.g.* sustained speed in GFlop/s). The **w-rand** StarPU scheduler takes it into account by randomly dispatching tasks onto the processing units with a probability proportional to the respective computational power. This avoids the situation of Figure 9 and brings much better load balancing at the end of the application completion. It however does not take into account that some tasks may *e.g.* perform really well on a GPU and actually tend to reach the theoretical computational power, while others do not perform so well, and should rather be scheduled on the CPU, leaving the GPUs to the former. More refined StarPU schedulers thus rather use models of the execution time of the different tasks instead of just the processing units.

5.3.1 Taking task duration into account

The HEFT scheduling algorithm (Heterogeneous Earliest Finish Time [23]), implemented in the **heft-tm** StarPU policy, is illustrated in Figure 10. It makes use of performance prediction to keep track of the expected dates $Avail\,(P_i)$ at which each processing unit will become available (after all the tasks already assigned to it complete). A new task T is then assigned to the processing

unit P_i that minimizes the new termination time with respect to the expected duration $\underset{P_i}{Est}(T)$ of the task on the unit *i.e.*

$$\min_{P_i} \left(Avail(P_i) + \underset{P_i}{Est}(T) \right)$$

5.3.2 Taking data transfers into account

The time to transfer data to *e.g.* accelerators is actually far from negligible compared to task execution times, and can sometimes even become a bottleneck. We therefore extended the **heft-tm** policy into the **heft-tmdp** policy which takes data locality into account thanks to the tight collaboration between the scheduler and the data management library presented in Section 4. In addition to the computation time, the scheduler computes a penalty based on the times $\mathcal{T}_{j \to i}(d)$ required to move each data d from P_j (where a valid copy of d resides) to P_i. Such penalty of course reduces to 0 if the target unit already holds the data, *i.e.* $j = i$. The resulting minimization is

$$\min_{P_i} \Big(\underbrace{Avail(P_i) + \underset{P_i}{Est}(T)}_{\text{termination time}} + \underbrace{\sum_{data} \min_{P_j} \left(\mathcal{T}_{j \to i}(data) \right)}_{\text{data penalty}} \Big)$$

5.3.3 Prefetching data

As explained in Section 4.2, hardware often allows data transfers to overlap with the actual computations, thus maximizing the time spent computing. This is exploited by StarPU by having the scheduler request from the DSM data transfers for a task as soon as the placement of that task gets decided. The DSM then queues the required data transfer orders to the various device drivers and the scheduler can reduce its estimation of the transfer time. This can be particularly important when using multiple accelerators which do not support direct transfer, in which case the data has to be written back to memory first, hence at least doubling the transfer time. The **heft-tm** and **heft-tmdp** policies have thus been extended into the **heft-tm-pr** and **heft-tmdp-pr** policies.

6 Predicting task completion time

We described how task scheduling can benefit from models that predict the performance of task execution and data transfer. Diamos *et al.* mention [9] that their HARMONY runtime could also benefit from such models. It is worth noting that in the case of StarPU, such models do not necessarily need to be extremely accurate, since scheduling is done at runtime and can compensate not-so-optimal decisions according to what actually happened. We now detail how StarPU automatically builds these models.

6.1 Task performance models

More or less automatic ways exist to build task performance models. An extreme approach is to have the programmer explicitly construct them. This however requires extra experimentation, as well as a lot of knowledge of both

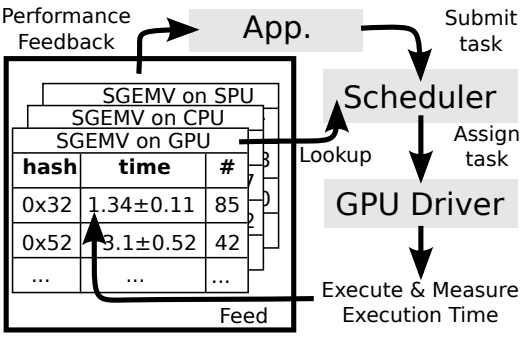


Figure 11: Performance feedback loop.

the algorithm and the underlying machine. Such work also needs to be repeated for each platform, due to details as simple as cache size, so that this is not a realistic approach in general.

For very common kernels and libraries (BLAS, typically), an important literature includes performance models for the main kernels. Using them is possible with some extra programming efforts to tune their parameters according to the architecture. These models are however sometimes pretty simplistic (*e.g.* limitations in the cache size), and offline precalibration often does not capture real-world use, where contention and cache sharing can severely impact performance.

In order to get performance models which actually match what can be observed with real applications, we use history-based prediction models. When a task is executed, StarPU measures its duration, and to schedule a task, it can consult the history to take into account the performance obtained during previous executions.

This assumes some regularity hypothesis: execution time should be independent from the content of data itself, thanks to the flow control being static for instance. This is accurate for the kernels mentioned above. When provided with their literature performance models, StarPU can thus tune the latter during the execution by regression analysis, according to the underlying architecture, but also according to the general contention impact of the application.

In addition to that, StarPU can use history-based prediction for kernels without literature performance model, by assuming that for a given application they are mostly always called with the same parameters, the typical task granularity for instance. Given task parameters characteristics (the size of the data, typically), StarPU computes a hash characterizing this task's complexity, and that can thus be used as an index in the history tables from which an average of the previous execution times can be obtained. Figure 11 illustrates the resulting feedback loop between these tables, the Scheduler, and the actual execution measurement The history is saved to a file to be re-used for further executions. More details can be found in a previous paper [1].

The key advantages of this approach is not only being efficient, but also being transparent for the programmer (he already has to specify the data size for transfers anyway), and it can thus be applied to any kernel which has regular execution times. When unsure about the latter property, StarPU also computes the standard deviation of the measurements and can even provide a histogram

```
1  while (machine_is_running())
2  {
3      task = pop_task();
4
5      fetch_task_input(task);
6      task->cl->cpu_func(task->interface, task->cl_arg);
7      push_task_output(task);
8
9      handle_task_termination(task);
10 }
```

Figure 12: Driver for one CPU core

of the distribution of execution times, so that the variability of performance can be checked. StarPU can also generate a detailed Gantt chart of the actual execution, which helps a lot in understanding performance, particularly on complex heterogeneous architectures for which interference can happen between several different parts of the application for instance..

6.2 Data transfer models

Compared to the internal memory bandwidth of accelerators (several hundreds of GB/s), the speed of the PCI bus (half a dozen GB/s) is a real concern, and data transfer time can become as important as the computation time itself and thus be critical for scheduling decisions. To estimate the time to transfer some data, StarPU simply uses an estimation of the latency and the asymptotic bandwidth between these two memory nodes, which can be either measured through offline experiments, or, better, inferred from online measurements through regression analysis.

This approach catches various architecture artifacts, like heterogeneity and topology of PCI buses (8x *vs.* 16x and NUMA effects), as well as asymmetry of transfer paradigms (upload and download speeds are usually different on CUDA devices).

7 Porting StarPU to various hardware

As a reminder of Figure 1, when a task is submitted by the application, it first has to wait for its dependencies, then gets dispatched by the scheduler to some driver, the DSM ensures that all the data are available and the driver eventually offloads the codelet. When the latter completes, tasks depending on it are released and a callback is executed.

Porting StarPU to a new kind of device actually boils down to implementing functions transferring data between the device and the main memory, and executing tasks on the accelerator. To achieve the latter, drivers run an execution loop. For instance, Figure 12 summarizes the loop of the driver for a CPU core. It continuously fetches tasks from the scheduler (`pop_task`), requests the DSM to make sure all data are available (`fetch_task_input`, which possibly calls transfer functions of other drivers to fetch data back to main memory), and actually calls the codelet. After codelet termination, data can be

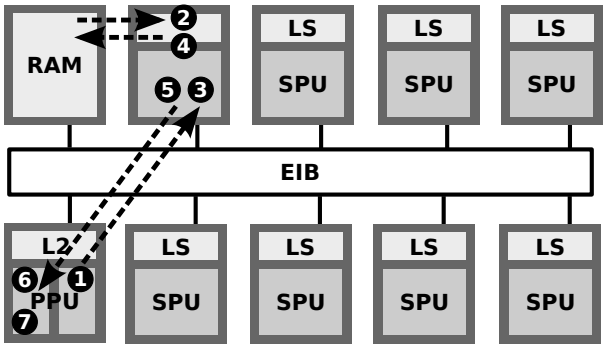


Figure 13: Offloading tasks with the Cell-RTL.

released (`push_task_output`) and tasks depending on this task can be released and the callback called (`handle_task_termination`). The driver work is thus completely decoupled from the scheduling decision and DSM operation.

7.1 Heterogeneous multi-GPU

The implementation of the driver for an NVIDIA GPU is relatively straightforward: data transfer functions use the API from NVIDIA and the driver can directly call the codelet's `cuda_func()` function[1] which makes standard CUDA or CUBLAS kernel calls. Some shortcomings of CUDA raise issues when handling multiple GPUs, but the design of StarPU overcomes them quite naturally.

Due to the CUDA specification and implementation, switching between the CUDA context of different GPUs is very costly. StarPU's CUDA driver thus runs one execution loop in a separate kernel thread for each GPU device (this is hence where the codelet's `cuda_func()` function is called). Because the CPU consumption of these threads is far from negligible, StarPU devotes CPU contexts (cores or hyper-threaded contexts) for them.

The CUDA API also does not permit to issue direct GPU-GPU memory transfers, they have to be achieved in two steps through the main memory. Thanks to its chained asynchronous data transfer described in Section 4.2, StarPU overcomes this by issuing the two transfer orders to the two driver instances.

7.2 The Cell B/E

The Cell B/E processor is a heterogeneous multicore chip composed of a main hyper-threaded core, named PPU (Power Processing Unit), and 8 coprocessors called SPUs (Synergistic Processing Units).

The StarPU port to the Cell [2] actually relies on the Cell-RTL (Cell Run Time Library) [18] to perform task offloading and manage data transfers between the main memory and local stores (LS) on the SPUs. Figure 13 depicts how the Cell-RTL works: when a task for some SPU is submitted on the PPU, the latter sends a message to the former (1); an automaton running on each SPU reads the message, fetches data into the LS (2), executes the corresponding task (3),

[1] in the case of Figure 6, the `cublas_SGEMM` function

commits the output data back to the main memory (4), and sends a signal to the PPU (5); when the latter detects this signal (6), a termination callback is executed (7). To greatly reduce the synchronization overhead, the Cell-RTL can submit chains of tasks to SPUs.

To fully exploit the possibilities of the Cell-RTL, the StarPU Cell B/E driver loop automatically builds chains of tasks before submitting them. Thanks to SPUs being almost full-fledged cores, the flexible Cell-RTL API permits to devote only one thread to submitting tasks to the different SPUs.

In a way, StarPU leverages the Cell-RTL by adding scheduling facilities and providing with the high-level data management and task dependencies enforcement, permitting efficient task chaining. StarPU could leverage other backends like IBM's ALF [8] or CellSs' runtime [6].

8 Evaluation

To validate our approach, we show that StarPU efficiently uses all the available processing units with a low overhead, and we analyze the performance of some numerical algorithms under different scheduling strategies. Eventually, we show that selecting the best granularity is a challenging issue on accelerator-based platforms.

8.1 Experimental testbed

To evaluate the performance of StarPU, we have used two platforms both running Linux 2.6 and CUDA 2.3:

A **homogeneous multi-GPU platform** composed of two X5550 (Nehalem) hyper-threaded **quad-core** processors running at 2.67 GHz with 48 GB of memory divided in two NUMA nodes. It is equipped with **3 NVIDIA Quadro FX5800** with 4 GB of memory.

A **heterogeneous multi-GPU platform** composed of a E5410 Xeon **quad-core** processor running at 2.33 GHz with 4 GB of memory. It is equipped with **an NVIDIA Quadro FX5800** with 4 GB of memory and **a: NVIDIA Quadro FX4600** with 768 MB of memory; the second GPU does not support asynchronous data transfers.

8.2 Distribution of the computation among heterogeneous platforms

Even with its simplest **greedy**, scheduler, StarPU simultaneously takes advantage of all available resources according to their respective computational power: Figure 14 (resp. 15) shows the proportion of tasks attributed to the different processing units in a multi-GPU system (resp. heterogeneous multi-GPU system). On the left-hand side, we have a blocked matrix-product, and on the right-hand side, we have a band-pass filter implemented using FFTW and CUFFT. In both benchmarks, the GPUs become relatively more efficient than the CPUs and thus get attributed more tasks when the granularity increases. Figure 15 illustrates that StarPU is able to distribute task onto different models of GPUs with respect to their respective speed: a Quadro FX5800 is given more tasks than a Quadro FX4600 which is much less powerful.

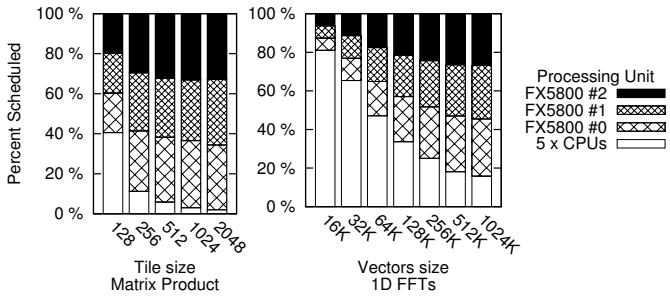


Figure 14: Distribution of the computation on a multi-GPU system.

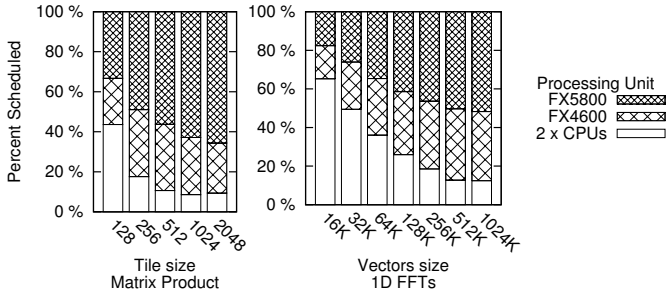


Figure 15: Distribution of the computation on a heterogeneous multi-GPU system.

Table 2: StarPU critical path overhead

	Task submission	Data Management	Execution
1 CPU	468 ± 7.3 ns	94 ± 8 ns	364 ± 17 ns
8 CPUs	470 ± 8.2 ns	113 ± 11 ns	733 ± 26 ns

8.3 Overhead evaluation

It is important to make sure that StarPU only has a minimal overhead compared to its potential gains. Table 2 gives the typical overhead measured during the execution of chains of one million empty tasks, and by repeating this experiment 128 times. We distinguish the overhead of task submission, data management (for a single piece of data), and the overhead of the task execution itself. The second line gives the performance when this experiment is performed on 8 CPUs which are all associated with only one single queue. Even in this extreme case the overhead remains low, which shows that StarPU properly handles highly contended situations with a typical overhead smaller than a few micro-seconds.

In comparison, Volkov and Demmel report that CUDA functions typically incur a 3 to 7 μs overhead for asynchronous launch, and 11 μs for synchronous calls [24]. Since data transfers are also taking tens of micro-seconds, we conclude that the overhead introduced by StarPU is acceptable with respect to the performance improvement resulting from scheduling policies.

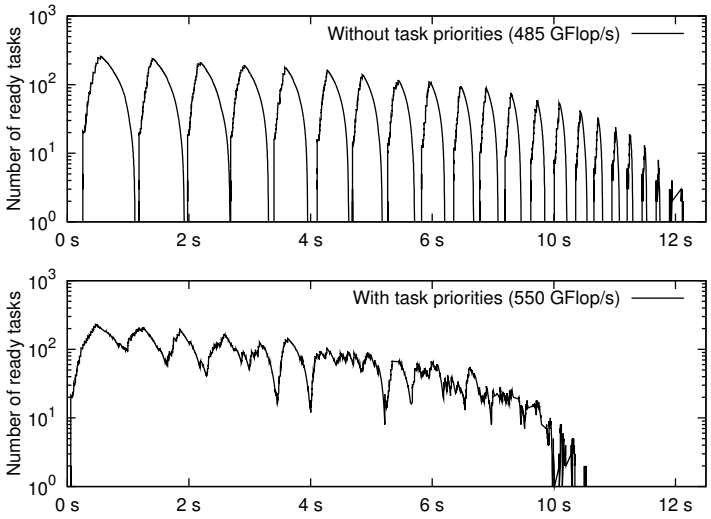


Figure 16: Impact of priorities on Cholesky decomposition.

8.4 A case study: LU and Cholesky decompositions

Implementing LU and Cholesky blocked decompositions for multicore machines equipped with multiple GPUs is straightforward with StarPU, even for problems that are larger than the memory available on the accelerators.

8.4.1 Prioritizing tasks

It is however crucial to maintain enough parallelism to feed such a massively parallel machine during the entire application. The LU and Cholesky decomposition algorithms are known to suffer from a lack of parallelism, that is usually avoided by performing critical tasks as soon as possible by the means of task priorities.

Figure 16 shows the evolution of the number of ready tasks during the execution of a Cholesky decomposition running on the homogeneous multi-GPU machine with our best scheduling strategy, **heft-tmdp-pr**. While every tasks has the same priority on the top curve, we put a maximum priority for the critical tasks in the second case. Priority-aware scheduling here prevents substantial loss of parallelism, so that we observe a typical 15 % speed improvement.

8.4.2 Modeling task performance

Figure 17 shows the speed of LU decomposition according to the problem size, and for various scheduling policies. The second bottom-most line gives the performance obtained by the **heft-tm** strategy with the support of the automatically tuned history-based performance models presented in Section 6.1. While no application code modification was required compared to the **greedy** policy, the **heft-tm** strategy significantly outperforms the simple greedy scheduling policy, especially for small problems which tend to have little parallelism. In

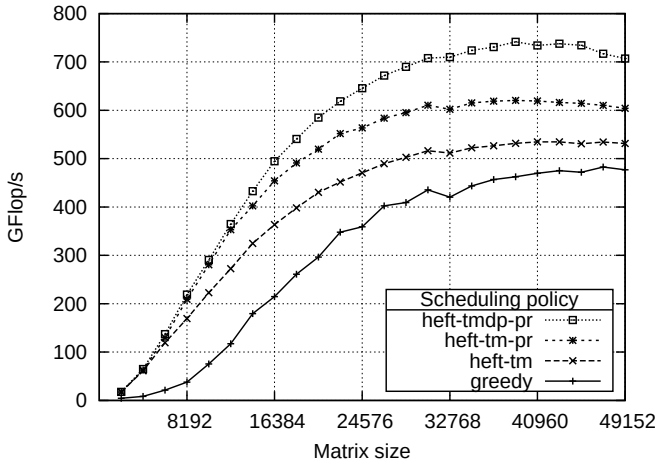


Figure 17: Impact of scheduling policies on LU decomposition.

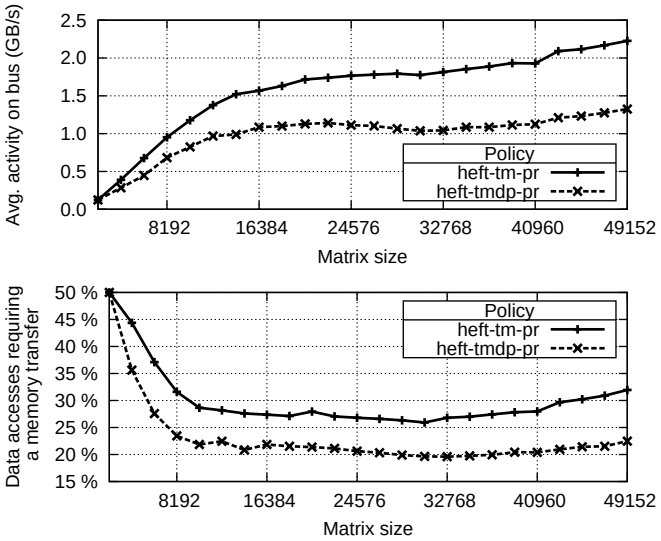


Figure 18: Penalizing non local data accesses reduces the total amount of data transfers by lowering the ratio of data accesses that require memory transfers.

these cases, it is indeed critical that the most efficient processing units (ie. GPUs) get the most critical tasks.

In a previous paper [3], we have shown that in the case of a quad-core machine equipped with a single GPU, this strategy obtains a super-linear efficiency: the processing power of the hybrid system is slightly higher than the sum of the powers of a GPU and of the CPUs, taken individually. This is possible because the LU decomposition is composed of different types of tasks, with different relative speedups between their CPU and their GPU implementations. A matrix product (GEMM) could be accelerated 20 times by a GPU, while the resolution of a triangular system (TRSM) may *only* be accelerated 10 times by a GPU. With

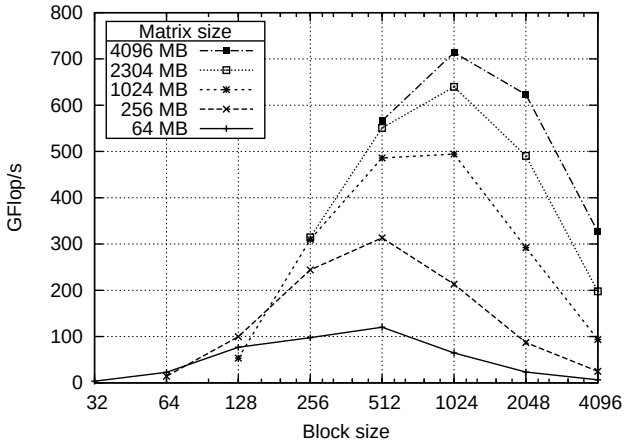


Figure 19: Impact of granularity on the efficiency of LU decomposition on GPUs+CPUs.

the **heft-tm** policy, the processing units therefore tend to receive the types of tasks that they most efficiently handle, and they process less tasks for which they are relatively inefficient. **StarPU thus actually takes advantage of the heterogeneous nature of accelerator-based platforms**.

8.4.3 Scheduling data transfers

Taking data transfers into account while scheduling tasks is critical. This is especially important for multi-accelerator platforms where memory buses are major bottlenecks.

The **heft-tm-pr** policy attempts to mask the cost of memory transfers by overlapping them with computations thanks to the prefetch mechanism described in Section 5.3.3. Figure 17 shows a typical 20 % speed improvement over the **heft-tm** strategy which does not prefetch data.

However, data prefetching does not reduce the total amount of memory transfers. In Section 5.3, we described how the **heft-tmdp-pr** strategy extends **heft-tm-pr** by penalizing non-local data accesses. The average activity on the bus is shown at the top of Figure 18: penalizing remote data accesses, and therefore favoring data locality has a direct impact on the total amount of data transfers which drops almost by half. Likewise, the bottom of Figure 18 shows that the processing units tend to work more locally. On Figure 17, this translates into another 15 % reduction of the execution time.

8.5 How about granularity?

In this section, we show that selecting the best granularity is not trivial, especially in the context of heterogeneous platforms, and we give some insights about how StarPU could handle this challenge.

Table 3: Dynamically adapted granularity for the LU decomposition of a 1GB matrix

Block size	(512x512)	(1024x1024)	hybrid
Time (ms)	5991 ± 29	5974 ± 57	5691 ± 83
Speed (GFlop/s)	489.4 ± 2.4	490.8 ± 4.7	515.2 ± 7.4

8.5.1 Making trade-offs to select the a granularity

Figure 19 shows that depending on the problem size, the granularity for which the best result is obtained can vary. Separately precalibrating the various computational kernels individually for the different architectures is thus not sufficient. Libraries relying on precalibration to detect optimal block sizes [26, 16] should therefore take into account the factors that influence the value of the optimal granularity.

Different factors may indeed affect the choice of the grain size in the context of heterogeneous machines equipped with one of more accelerators. The relative efficiency of a task on various architectures depends on the data size: GPUs are for instance large vector systems that are only efficiently used when all the vector processors are used simultaneously, which is not possible with tiny data input. On the one hand, a small grain size means that there is a lot of small tasks, which potentially leads to a high overhead. On the other hand, too big a grain size causes processing units to become idle due to lack of parallelism.

Selecting the most appropriate granularity is therefore a matter of trade-offs when dealing with heterogeneous platforms equipped with accelerators.

8.5.2 Dealing with multiple grain sizes

As a result, while there can be an optimal grain size for a specific problem on a homogeneous machine, it may be that the best solution in a heterogeneous context is a mixture of multiple granularities.

A first simple approach is to explicitely decompose an application into multiple phases with different granularity, according to the evolution of the behavior of the computation. In the case of LU decomposition, we observe that there is a lack of parallelism at the end of the algorithm. Intuitively, we would like to offload tasks with a big granularity at the beginning of the algorithm to get goot performance on GPUs, and we should reduce the grain size when the amount of parallelism becomes too low to avoid load imbalance. Table 3 gives the speed of an LU decomposition on a 1 GB problem with the **heft-tmdp-pr** strategy. The first two columns gives the speed obtained with either (512×512) or (1024×1024) block sizes. The last column gives the performance obtained by starting with blocks of (1024×1024), and by reducing the grain size to (512×512) at the middle of the algorithm execution. The performance improvements by this approach which dynamically adapts the granularity are promising: the hybrid solution gets better performance than the best performance obtained with a single grain size (see the 1 GB curve of Figure 19 which does not reach 500 GFlop/s).

Besides, it is possible to use multiple granularities at the same time, either explicitly or implicitly. Programmers could submit tasks with multiple grain size by hand, but this requires significant programming efforts. This approach is applicable for applications which explicitly select the tasks that should be

executed on the accelerators, such as MAGMA [22], but such static schedules are hardly conceivable in a portable fashion. Another solution is to have *dividable* tasks: when StarPU detects that the granularity is not adapted, it could dynamically perform some local transformations on the graph of tasks, to either divide or merge tasks (similarly to the task chaining mechanism that we use on the Cell).

The difficulty of this implicit method is that StarPU would have to automatically make appropriate decisions about when to change the grain size, and how to select a better granularity. This could be achieved with performance feedback mechanisms, either offline or online. A *post-mortem* analysis would help in choosing the optimal block size or in detecting the different phases of a regular algorithm. A dynamic approach could use the metrics available from the scheduling engine, such as load imbalance, in collaboration with hardware performance counters (*e.g.* to detect a sudden contention).

9 Related Work

Recent years have seen the democratization of accelerator-based computation. A vast majority of the interests has however been devoted to writing efficient code for the accelerators. Relatively few projects have investigated the problem of tasks scheduling. To our knowledge we are the first to study this specific aspect in depth while offering a portable and high-level platform to experiment with task scheduling on such heterogeneous machines equipped with accelerators.

Various runtime systems have been designed (or adapted) to support heterogeneous multicore systems. Most of them are however offering an interface to offload tasks on only a single type of accelerator. While those interfaces hide most low-level technical issues, they also typically require the programmer to decide where the computation should take place, so that some knowledge of the underlying platform is still required.

Jimenez et al. [14] consider the problem of scheduling computations on multicore machines equipped with accelerators. However, contrary to StarPU which schedules tasks at a fine granularity, they only dispatch whole applications with respect to the relative speedups observed on the different processing units during previous executions, regardless of the data input. The burden of data management is also left to the programmer.

The **Charm++** runtime system was also implemented for both GPUs [25] and Cell processors [15] in a cluster environment. Even though the Offload API they proposed in [15] was adopted by most task-based approaches, there is currently no performance evaluation available yet, to the best of our knowledge.

Similarly to the low-level interface of OpenCL, **IBM ALF** [8] provides an interface to offload tasks onto heterogeneous platforms, such as the RoadRunner. ALF therefore supports both x86 and Cell processors. While ALF relieves programmers from numerous low-level concerns, it only offers very basic load balancing mechanisms thanks to the DaCS framework, and it is done at the process level contrary to StarPU which actually schedules tasks.

Likewise, the **Cell Run Time Library** (or Cell-RTL) [18] provides a convenient interface to offload tasks onto the SPUs of a Cell processor. As we have shown in Section 7.2, StarPU and the Cell-RTL are very complementary on this platform. The Cell-RTL indeed provides low-level Cell-specific optimiza-

tions (*e.g.* very efficient synchronization mechanisms) but it only has basic load balancing mechanisms which StarPU can replace. Cell-RTL also handles data movements within a Cell chip, but StarPU provides the higher-level abstractions required to manage them consistently.

Diamos *et al.* [9] present a "collection of techniques" for the **Harmony** runtime system which they "would like 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. Harmony however does not allow user-provided scheduling strategies contrary to StarPU which makes it possible to select the most appropriate policy at runtime. 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.* [4]. 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 [5], CellSs [6]). Some efforts are done to combine those different runtime systems: Planas *et al.* allow programmers to include CellSs tasks within SMPSs tasks [20], 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.

GPUSs permits to use multiple GPUs [5], but its load balancing mechanisms and data management are still very simple. **CellSs** also consider advanced task scheduling mechanisms taking data locality into account [7]. Contrary to the Cell-RTL (which we used to offload StarPU tasks onto SPUs), the runtime system of CellSs greatly benefits from software data caching which permits to directly direct transfers between SPUs. Adding this feature into Cell-RTL or using CellSs within StarPU's Cell driver would significantly improve our support for the Cell processor. Likewise, we plan to use StarPU as a target of the source-to-source compiler from StarSs.

The approach of the **Anthill** runtime system is very similar to StarPU. Teodoro *et al.* experimented [21] 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 Section 5.2. The tight integration of data management and task scheduling within StarPU makes it possible to tackle the problem of task scheduling onto multiple GPUs while taking data movements overhead into account, while Anthill only considers relative speedups to select the most appropriate processing unit.

10 Conclusion and Future Work

The StarPU runtime system and the experiments presented in this paper can be downloaded at `http://runtime.bordeaux.inria.fr/StarPU/`. The significant performance improvements that we exhibit on top of already optimized

computation kernels show that task scheduling and efficient code generation are orthogonal problems: producing the most efficient computational kernels is wasteful if those are not scheduled judiciously. We advocate that such scheduling techniques are required to cope with the growing number of accelerators in a scalable fashion: StarPU massively reduces the amount of data transfers without any application code modification.

The key contribution of StarPU is the tight collaboration between its high-level data management library and its portable scheduling engine which permits to easily design powerful scheduling policies. StarPU unlocks the portability of performance on complex accelerator-based platforms: it is for instance generic enough to transparently handle heterogeneous multi-GPU setups by hiding both low-level heterogeneity and by dispatching tasks according to the capabilities of the different units.

StarPU is not limited to multicore machines equipped with GPUs and Cell processors. Its asynchronous event-driven design will for instance make it straightforward to implement an OpenCL backend. The Larrabee processor should also benefit from the techniques adopted to exploit the Cell processor. Likewise, the dynamic code loading mechanism that we implemented in the Cell StarPU driver would be applicable to reconfigurable architectures such as FPGAs. StarPU would also be helpful in the case of fixed-functions hardware such as multicore DSPs which also raise heavy task and data scheduling requirements.

In the future, we plan to use StarPU as a backend for compiling environments (such as StarSs) which would generate StarPU tasks. We intend to improve our performance models with regards to the actual hardware: first by taking NUMA effects into account when scheduling tasks, and then by considering the non-uniform performance of the IOs on such NUMA platforms [17]. We will experiment with better data prefetching and data reclaiming policies to improve the performance in the case of out-of-core computations and data streaming applications. The importance given to the penalty for non-local data accesses should also be dynamically throttled depending on the load balance conditions. Similarly, the scheduling engine should be assisted by some execution feedback (*e.g.* from the hardware performance counters). Eventually, having different instances of StarPU collaborating through MPI would allow to exploit clusters of machines equipped with accelerators. Granularity is a key issue that we think has yet been ignored in the context of accelerator-based computation until now: we believe that our proposition of *dividable* tasks is an interesting basis to attack this challenging problem.

More generally, we envision StarPU as a meeting point to leverage the efforts in various research domains: it stands as a powerful backend for compiling environments (eg. StarSs), it offers a high-level playground to experiment with algorithms from the scheduling literature on actual platforms, and it provides portable performance to HPC libraries such as MAGMA.

Acknowledgments

This work has been supported by the ANR through the COSINUS (PROHMPT ANR-08-COSI-013 project) and CONTINT (MEDIAGPU ANR-09-CORD-025) programs. We thank NVIDIA and NVIDIA's Professor Partnership Program for their hardware donations.

References

[1] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *Proceedings of the International Euro-Par Workshops 2009, HPPC'09*, Lecture Notes in Computer Science, Delft, The Netherlands, August 2009. Springer.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Maik Nijhuis. Exploiting the Cell/BE architecture with the StarPU unified runtime system. In *SAMOS Workshop*, Lecture Notes in Computer Science, Samos, Greece, July 2009.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André 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.

[4] Eduard Ayguade, Rosa M. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, Josep M. Perez, and Enrique S. Quintana-Ortí. A proposal to extend the openmp tasking model for heterogeneous architectures. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP*, pages 154–167, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th Euro-Par Conference*, Delft, The Netherlands, August 2009.

[6] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. Exploiting Locality on the Cell/B.E. through Bypassing. In *SAMOS*, pages 318–328, 2009.

[7] Pieter Bellens, Josep M. Pérez, Felipe Cabarcas, Alex Ramírez, Rosa M. Badia, and Jesús Labarta. Cellss: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.

[8] Catherine H. Crawford, Paul Henning, Michael Kistler, and Cornell Wright. Accelerating computing with the cell broadband engine processor. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 3–12, New York, NY, USA, 2008. ACM.

[9] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200, New York, NY, USA, 2008. ACM.

[10] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment, 2007.

[11] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex

Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[13] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[14] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *HiPEAC*, pages 19–33, 2009.

[15] D. Kunzman, G. Zheng, E. Bohm, and L. V. Kalé. Charm++, Offload API, and the Cell Processor. In *Proceedings of the PMUP Workshop*, Seattle, WA, USA, September 2006.

[16] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *ICCS (1)*, pages 884–892, 2009.

[17] Stéphanie Moreaud and Brice Goglin. Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines. In *The 19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007)*, Cambridge, Massachussetts, November 2007.

[18] Maik Nijhuis, Herbert Bos, Henri E. Bal, and Cédric Augonnet. Mapping and synchronizing streaming applications on cell processors. In *HiPEAC*, pages 216–230, 2009.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 03 2007.

[20] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierchical task based programming with StarSs. *International Journal of High Performance Computing Application*, 23:284, 2009.

[21] George Teodoro, Rafael Sachetto, Olcay Sertel, Metin Gurcan, Wagner Meira Jr., Umit Catalyurek, and Renato Ferreira. Coordinating the Use of GPU and CPU for Improving Performance of Compute Intensive Applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, New Orleans, LA, September 2009. IEEE Computer Society Press.

[22] S. Tomov, J. Dongarra, and M. Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. Technical report, January 2009.

[23] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002.

[24] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[25] Lukasz Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois, 2008. http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

[26] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

Centre de recherche INRIA Bordeaux – Sud Ouest
Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex (France)

Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399