HAL
open science

Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures

Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, Bruno Raffin

To cite this version:

Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, Bruno Raffin. Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures. 6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG), Jan 2013, Berlin, Germany. hal-00780890

HAL Id: hal-00780890

https://inria.hal.science/hal-00780890v1

Submitted on 24 Jan 2013

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

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

Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures

Thierry Gautier[3], João V. F. Lima[12],
Nicolas Maillard[1], and Bruno Raffin[3]

[1] Federal University of Rio Grande do Sul (UFRGS), Brazil
[2] Grenoble University, France
[3] INRIA, Grenoble, France

thierry.gautier@inrialpes.fr, Joao.Lima@imag.fr,
nicolas@inf.ufrgs.br, Bruno.Raffin@inria.fr

Abstract. Most recent HPC platforms have heterogeneous nodes composed of a combination of multi-core CPUs and accelerators, like GPUs. Scheduling on such architectures relies on a static partitioning and cost model. In this paper, we present a locality-aware work stealing scheduler for multi-CPU and multi-GPU architectures, which relies on the XKaapi runtime system. We show performance results on two dense linear algebra kernels, Cholesky (POTRF) and LU (GETRF) factorization, to evaluate our scheduler on a heterogeneous architecture composed of two hexa-core CPUs and eight NVIDIA Fermi GPUs. Our experiments show that an online locality-aware scheduling achieve performance results as good as static strategies, and in most cases outperform them.

1 Introduction

With the recent evolution of processor design, future generations of processors will contain hundreds of cores. The success of these machines will rely on the ability to schedule the workload at runtime, even for small problem instances.

Several libraries [17,6] or languages such as Cilk [9], X10, Fortress, Chapel or OpenMP are designed to improve productivity with parallel constructs, such as *for each*. The main drawback of a *for each* construct is the addition of strong synchronization points to enforce the completion of a set of independent tasks, and the associated memory update, before a new task set can be executed. On the other hand, the data-flow model simplifies programming by unfolding parallelism based on data-flow dependencies between tasks. Runtime systems with support to data-flow programming are nowadays *de facto* standard for parallel linear algebra libraries on multi-cores [8,14,16]. Besides, they can automatically move data between address spaces, such as on multi-CPU or multi-GPU architectures [3,4,13,15] or clusters [7,10,18].

However, the data-flow software cited above have some restrictions for heterogeneous architectures. Since their programming models only consider one level of parallelism, without the capacity to adapt the computation to the resource,

their scheduling algorithms rely on static partitioning and cost models to guarantee the performance [3,14], and none support recursive tasks, except [7,15]. For instance, the ETF heuristic [5] used in [3] relies on a cost model to predict performance based on the application's tasks and the hardware resources. The uncertainties related to the cost estimations and a poor worst case when communication costs are high, may impact the overall performance.

In this paper, we introduce a locality-aware work stealing scheduler for multi-CPU and multi-GPU architectures. Our solution relies on XKaapi, a runtime system for data-flow task programming on heterogeneous architectures. XKaapi combines a C++ interface for data-flow programming (Section 2) and a work stealing based scheduler to support multi-CPU and multi-GPU architectures (Section 3). The main contributions we propose are:

- A locality-aware work stealing algorithm based on heuristics to manage data locality and tackle the cache-unfriendly problem of classic work stealing [12], which is critical on multi-GPU systems;
- A fully asynchronous task execution strategy on GPUs to overlap data transfers with GPU kernel executions.

We evaluate the XKaapi runtime with two dense linear algebra algorithms in double precision: Cholesky and LU factorization (Section 5). Our experiments show that XKaapi achieves a speedup of about 6 for Cholesky and 4.54 for LU when using eight GPUs and four CPUs over one GPU and one CPU. In terms of raw performance, with matrices of size 40960×40960, we attain about 1.79 TFlop/s on double precision Cholesky (3.92 TFlop/s on single precision), and 816.50 GFlop/s on double precision LU (1.24 TFlop/s on single precision).

2 Data-Flow Task Programming with XKaapi

The XKaapi[4] task model [11], as in Cilk [9], Intel TBB [17], OpenMP-3.0 or StarSs [4,7], enables non-blocking task creation: the caller creates the task and proceeds with the program execution. The semantic remains sequential such as XKaapi's predecessors Athapascan [10] and KAAPI [11], which was specialized for multi-CPU/multi-GPU iterative applications [13]. Still, in this paper, we introduce a general scheduling algorithm for multi-CPU/multi-GPU systems that enforces a locality-aware work stealing (Section 3).

XKaapi has several APIs (C, Fortran, C++) to program heterogeneous parallel architectures. In this paper, code fragments are presented using the C++ API. A XKaapi program is a sequential code complemented with annotations or runtime calls to create tasks. Parallelism is explicit, while the detection of synchronizations is implicit [11]: the dependencies between tasks and the memory transfers are automatically managed by the runtime. A task is a side-effect free function call that returns no value except through its effective parameters. Tasks are created by calling the template function `ka::Spawn`.

[4] `http://kaapi.gforge.inria.fr`

```
/* left looking Cholesky factorization */
for( k=0; k < N; k+= blocsize ) {
  ka::Spawn<TaskPOTRF>()( A(rk,rk) );
  for( m=k+blocsize; m < N; m+= blocsize)
    ka::Spawn<TaskTRSM>()( A(rk,rk), A(rm,rk) );
  for( m=k+blocsize; m < N; m+= blocsize) {
    ka::Spawn<TaskSYRK>()( A(rm,rk), A(rm,rm) );
    for( n=k+blocsize; n < m; n+= blocsize )
      ka::Spawn<TaskGEMM>()( A(rm,rk), A(rn,rk), A(rm,rn) );
  } }

/* Signature defines task parameters and access modes */
struct TaskSYRK: public ka::Task<2>::Signature<
    ka::R<ka::range2d<double> >,
    ka::RW<ka::range2d<double> > >{};

template<> struct TaskBodyCPU<TaskSYRK> { /* CPU version */
  void operator( ka::range2d_r<double> A,
                 ka::range2d_rw<double> C )
  { cblas_dsyrk( A→dim(0), A→dim(1), A→ptr(), A→ld(), C→ptr(), C→ld() ); } };

template<> struct TaskBodyGPU<TaskSYRK>{ /* GPU version */
  void operator( ka::gpuStream stream,
                 ka::range2d_r<double> A,
                 ka::range2d_rw<double> C )
  { cublasDsyrk( kaapi_cublas_handle(stream),
      A→dim(0), A→dim(1), A→ptr(), A→ld(), C→ptr(), C→ld() ); } };
```

Fig. 1. Example of a XKaapi C++ Cholesky factorization. It shows a task *Signature* with its parameters and access modes, as well as CPU and GPU implementations. The calls to `cblas_dsyrk` and `cublasDsyrk` are simplified.

The code fragment of Figure 1 illustrates how to program a Cholesky factorization using the C++ API. The `ka::Spawn<Task>` creates a task of type `Task`. Each parameter `rk,rm,rn` corresponds to a range of indexes, and a construction such as `A(rm,rk)` represents the sub-matrix of elements `A(i,j)` where `i,j` are in the range `rm,rk`. The data type `range_2D` is an abstraction to view a memory region as a 2D array. Figure 1 also illustrates the definition of a task *Signature* (`TaskSYRK`) that includes the task parameters and their access modes (read `R` and/or write `W` and/or concurrent write `CW`). The implementations for CPU and GPU are given by the specialization of the `TaskBodyCPU` and `TaskBodyGPU` template class, respectively.

At least one implementation is expected per task signature (`TaskSYRK` in the example). The implementation can be recursive, calling the task signature and leaving to the scheduler the freedom to choose the more relevant implementation.

3 Extension for Multi-CPU and Multi-GPU

This section describes the features to support multi-CPU and multi-GPU[5] in XKaapi through asynchronous task execution, concurrent GPU operations as provided by recent Fermi GPUs, and software cache memory.

3.1 Asynchronous Task Execution

Once a task is selected, the runtime ensures consistency of its input data on the GPU device before the GPU kernel executes. The runtime assumes that the GPU task implementation launches the GPU kernels asynchronously. Once a task implementation has launched computations on a GPU, the scheduler starts the execution of the next selected task by sending its input data in advance. This enables to overlap data transfers with kernel executions.

We empirically found that the best performance gain is obtained when having two tasks being processed per GPU. Starting more tasks do not increase performance significantly and reduce the capacity to balance the work load, because tasks can not be aborted neither reactivated after the start of a GPU transfer.

Data transfers and kernel invocation on a GPU are handled asynchronously as well as the completion of these operations. We have gathered these functionalities in an extension of CUDA streams presented in the next section.

3.2 Concurrent GPU Operations

Recent GPUs, such as NVIDIA's Fermi and Kepler, support new features for asynchronism. For instance, Fermi GPUs have one execution engine and two copy engines, enabling to concurrently perform a kernel execution and memory transfers (two-way host-to-device and device-to-host), under the condition that no explicit nor implicit synchronization occurs.

We developed a mechanism to take advantage of this asynchronism for multi-GPU systems. XKaapi splits the execution of a GPU task in two basic operations: host-to-device input transfers (H2D); and `TaskBodyGPU` execution (*i.e.* launch of CUDA kernels) (K). Since concurrency between data transfers and kernel launches must use CUDA streams, we defined a new data structure, called *kstream*, that groups together three CUDA streams: a stream for host-to-device transfer, a stream for kernel execution and a stream for device-to-host transfer. Once the *kstream* detects the event completion, it calls the callback function with its argument as parameter.

3.3 Data Management and Software Cache

XKaapi manages GPU memory through a software cache, based on the Least Recently Used (LRU) replacement policy. Each GPU thread maintains a FIFO queue of allocated memory blocks. When a GPU task requires accessing a host

[5] Our current version supports NVIDIA CUDA.

memory block that is not present on the GPU, the runtime will allocate memory and insert it in its own queue. In order to enable asynchronous memory transfers with CUDA, user data is page-locked through specific CUDA library function (`cudaHostRegister`).

If its memory is full, a GPU tries to evict the least recently used memory block of its own queue (LRU policy). If possible, unused blocks are reused without being freed. This optimization avoids unnecessary CUDA calls.

Consistency is guaranteed by a lazy strategy using a write-back policy. Data transfers to or from the GPU occur only when a task accesses data and when the data is in an invalid state in the target address space. This policy avoids unnecessary transfers, unlike write-through policy [16,3,7]. All transfer operations are asynchronous and rely on the use of our *kstream* data structure to signal the completion of operations.

4 Locality-Aware Work Stealing

We extend each CPU or GPU thread with a local queue named *mailbox* in which remote threads can push tasks. This is similar to the approach proposed in [1], but without explicit locality annotation. Our locality-aware work stealing pushes the successors of a task to selected remote resources (CPU or GPU) based on meta-data information attached to each user data.

We developed two heuristics for local optimization, called **H1** and **H2**, using these meta-data:

H1 : For each task to be activated, XKaapi first goes through every task input parameter and looks for the resources where the parameter is valid, and its size. The resource which owns the biggest sum of input bytes in valid state is then chosen as the host to run the task.

H2 : This second heuristic is based on the data access modes. It tries to reduce the invalidations of the data replicas: the scheduler pushes a newly activated task on the resource's mailbox that has a valid copy of its *write* or *exclusive* accessed parameters. If more than one resource is eligible, then the scheduler simply selects a resource at random among the set of eligible ones.

The XKaapi work stealing algorithm polls each time a GPU thread is idle. The callback mechanism enables to compose a sequence of operations and it is typically used by the GPU work stealing algorithm, first to insert data transfers for the input of a task, and then to invoke the kernel launch when the transfer ends. Experiments show that the heuristic *H2* usually makes better local decisions, except for embarrassingly parallel applications, such as matrix product, where they both lead to a similar performance.

5 Experiments

All experiments have been conducted on an heterogeneous, multi-GPU system, named "Idgraf". Idgraf is composed of two hexa-core Intel Xeon X5650 CPUs

(12 CPU cores total) running at 2.66 GHz with 72 GB of memory. It is enhanced with eight NVIDIA Tesla C2050 GPUs (Fermi architecture) of 448 GPU cores (scalar processors) running at 1.15 GHz each (2688 GPU cores total) with 3 GB GDDR5 per GPU (18 GB total). The machine has four PCIe switches to support up to eight GPUs. When two GPUs share a switch, their aggregated PCIe bandwidth is bounded to the one of a single PCIe 16x. Experiments using up to four GPUs always use one GPU per PCIe switch to avoid this bandwidth constraint. On the other hand, experiments using more than four GPUs have to share some pairs of GPUs through the PCIe switch.

We used as software environment GNU/Linux Debian *squeeze* x86/64, the compiler GCC 4.4, CUDA 4.1, and the library ATLAS 3.9.39 for the CPU versions of BLAS and LAPACK.

5.1 Dense Linear Algebra Benchmarks

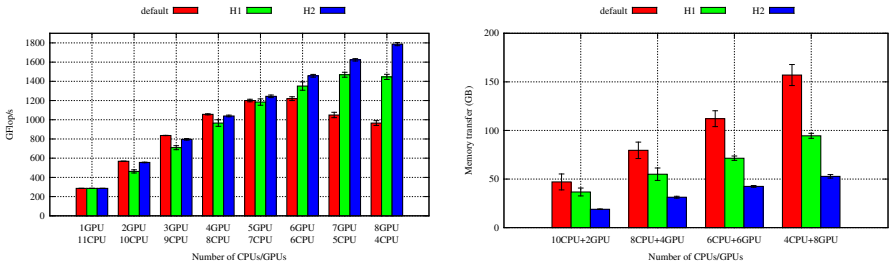
Our experiments use the parallel version of the dense linear algebra problems Cholesky and LU factorization, as found in PLASMA [2,8]. The algorithms have been re-implemented in XKaapi to use its low overhead task creation. The matrix data layout is the same as in PLASMA (tile data layout). The parallel Cholesky factorization is a two levels parallel algorithm: at the upper level, we use the PLASMA algorithm with 1024×1024 tiles; at the lower level the panel Cholesky factorization (`DPOTRF` or `SPOTRF`) is parallelized using the same parallel algorithm as at upper level by decomposing one tile in sub-tiles of size 128×128. Our LU factorization is based on four PLASMA kernels: `GETRF`, `GESSM`, `TSTRF`, and `SSSSM`. We implemented the update tasks (`GESSM` and `SSSSM`) for GPUs based on PLASMA and MAGMA, and we used the panel tasks (`GETRF` and `TSTRF`) CPU kernels from PLASMA. The LU tile size is 1024×1024. We have not used autotuning to select the sizes of the tile and sub-tile, but an empirical approach: after a few experiments showing their average good performances, we have decided to use theses values.

Each result is a mean of 30 executions. The 95% confidence interval is represented on the graphs.

5.2 Comparison of Work Stealing Heuristics

In this section, we compare the performance of the $H1$ and $H2$ heuristics (see section 4) against the default work stealing algorithm (label *default*), on the Cholesky (`DPOTRF`) and LU (`DGETRF`) factorization. The matrix size is constant (40960×40960) while we vary the number of GPUs. In addition to GPUs, we involve in the computations all remaining CPU cores, out of the 12 available, after removing the ones each GPU monopolizes to run its GPU thread.

Cholesky factorization Figures 2(a) and 2(b) illustrate our performance results. We conclude that: (a) the default heuristic has a bigger communication footprint that explains its poor scalability on more than four GPUs; (b) heuristic

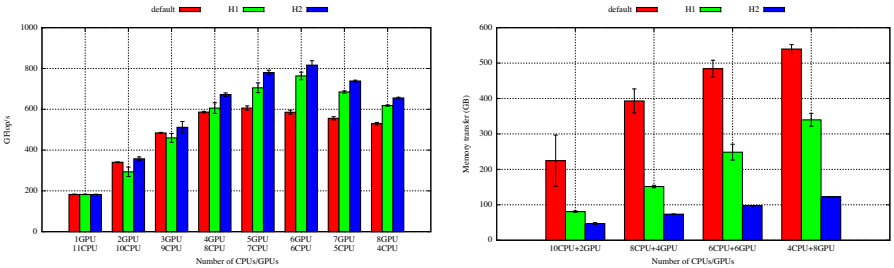


(a) Performance in GFlop/s. (b) Total data transfers (GB).

Fig. 2. Performance results of `DPOTRF` on eight GPUs and four CPUs for a matrix size of 40960×40960.

$H1$, which reduces the communication volume, enables a gain in scalability up to six GPUs; (c) heuristic $H2$ has the lowest volume of data transfers and scales up to eight GPUs. The peak performance with $H2$ is 1.79 TFlop/s in double precision and 3.92 TFlop/s in single precision.

LU factorization Figures 3(a) and 3(b) illustrate our performance results. In a similar way, heuristics $H1$ and $H2$ reduce communication volume. The peak performance with $H2$ is 816.50 GFlop/s in double precision and 1.24 TFlop/s in single precision. However, they scale up to six CPUs and six GPUs. Since LU has more CPU kernels than Cholesky, the scheduling heuristics has less impact on raw performance.



(a) Performance in GFlop/s. (b) Total data transfers (GB).

Fig. 3. Performance results of `DGETRF` on eight GPUs and four CPUs for a matrix size of 40960×40960.

Conclusion The local optimization decisions made by the heuristics do not ensure global reduction of data transfers. The second heuristic $H2$ tries to minimize

cache invalidations and seems to be more interesting: its effect is to keep data local to the resources, applying the classical "owner compute rule". The gain here is that the runtime automatically computes the right device to schedule the tasks without any programmer annotation.

Besides, it is important to optimize the execution of tasks on the critical path such as the panel tasks in Cholesky and LU. Our Cholesky implementation with two-level parallelism allows the use of all remaining CPU cores for panel tasks and reduces idle time of GPUs with update tasks. On the other hand, our LU version with two panel tasks benefits from more CPUs than GPUs. Hence, in our experiments, LU scales better with up to six CPUs and six GPUs than four CPUs and eight GPUs.

In experiments with more than four GPUs, at least two GPUs share the same PCIe-16x bus. Consequently, a scheduling algorithm that introduces a lot of memory transfer is more penalized on such hardware.

We note that our heuristic allows to obtain very good results. To our knowledge, this is the first time that teraflop performances are reported on a multi-core machine with up to eight GPUs. Moreover, these results were obtained using a purely dynamic work stealing algorithm.

5.3 Overlapping on Multi-GPU

Figure 4 refines the analysis of the performance impact when data transfers are overlapped with kernel executions, with the default work stealing and the $H2$ heuristic, on the Cholesky factorization. We have used We can also note

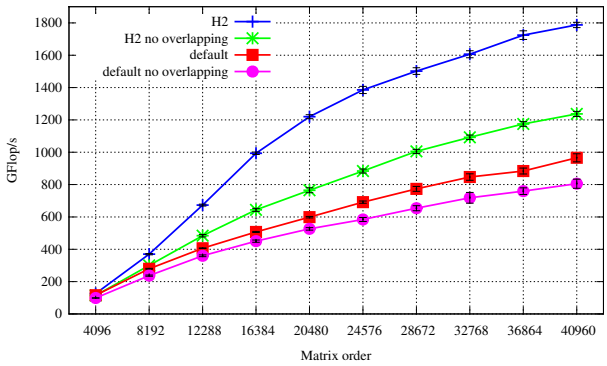
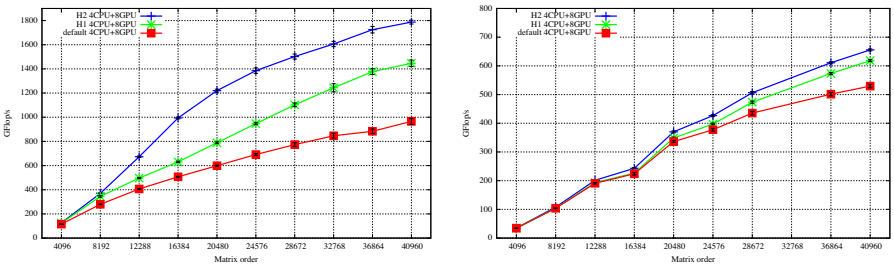


Fig. 4. Gain of overlapping on `DPOTRF` using 4 CPUs and 8 GPUs.

that the performance gain between the default work stealing four CPUs and eight GPUs for this experience, and a varying matrix size. With the default work stealing strategy, the overlap enables to increase the performance of up to 160.28 GFlop/s for the largest matrices (40960×40960). For small matrices the



(a) Cholesky. (b) LU.

Fig. 5. Scalability of the two heuristic and default work stealing with `DPOTRF` (left) and `DGETRF` (right) on eight GPUs and four CPUs.

difference is of course negligible. With the $H2$ heuristic, the gain reaches 550.48 GFlop/s of difference for 40960×40960. and our $H2$ heuristic is about 431.45 GFlop/s without any overlapping. Hence, it supports the importance of our $H2$ heuristic even without other GPU optimizations.

5.4 Scalability of the Cholesky and LU Factorizations

Figure 5 gives an overview of the performances that have been achieved on the Cholesky and LU factorizations for difference matrix sizes on eight GPUs and four CPUs using our two heuristics and default work stealing. Except for matrices of size 4096×4096, which results are almost equal, $H2$ gives the best performance for all matrix sizes and scales as the matrix size grows.

6 Related Works

OmpSs [7] is a programming tool that provides a set of OpenMP-like pragmas and a runtime system to schedule tasks while preserving dependencies. OmpSs locality-aware scheduling is similar to our $H1$ heuristic. To our knowledge, OmpSs has concurrent execution and data transfers in GPUs, but it shows some issues with matrix sizes that can not be entirely stored into the GPU memory.

StarPU is a runtime system for scheduling a DAG of tasks on heterogeneous architecture [3]. In a similar way, StarPU provides a programming model for heterogeneous architectures and exposes an API to describe a scheduling policy that allows flexibility in the work distribution. Its scheduler uses the HEFT [5] algorithm to schedule all ready tasks, thanks to cost models for data transfer and task execution. In a recent work [15], we show that our dynamic work stealing without heuristic (labelled 'default' in experimental section) reaches the same level of performances as StarPU on the matrix product and Cholesky factorization with a sequential panel factorization.

In the context of dense linear algebra algorithms, PLASMA [8] provides fine-grained parallel linear algebra routines with dynamic scheduling through QUARK, which was conceived specially for numerical algorithms. FLAME [16] is a high-level notation to express algorithms for dense linear algebra operations on multi-CPU/multi-GPU. MAGMA [19] implements static scheduling for linear algebra algorithms on heterogeneous systems composed of GPUs. Recently it has included some methods with dynamic scheduling in multi-CPU and multi-GPU on top of QUARK or StarPU, in addition to the static multi-GPU version.

Our $H2$ heuristic is inspired from [1], but with an automatic scheme to (locally) reduce the number of cache invalidations instead of the explicit annotation of the user code. In SLAW [12] a similar heuristic is experimented. As in [1], the programmer is responsible to explicitly specify the location where his task need to run.

To the knowledge of the authors, this is the first time that results on classical linear algebra subroutine are reported with more than four GPUs. Moreover, almost all previous reported results with high level of performances on a multi-GPU/multi-CPU are based on static scheduling.

7 Conclusion

In this paper, we presented a locality-aware work stealing scheduler for multi-CPU and multi-GPU architectures, which relies on XKaapi, a runtime system for data-flow task programming on heterogeneous architectures. XKaapi enables dynamic scheduling based on work stealing for multi-CPU and multi-GPU architectures. The key contributions of this paper include (1) an original locality-aware work stealing for multi-GPU systems based on local reduction of cache invalidations, (2) a fully asynchronous task execution strategy on GPUs to overlap transfers with kernel executions.

Our experiments report results up to eight GPUs on two dense linear algebra problems. The performance results obtained are about 1.79 TFlop/s for Cholesky and 816.50 GFlop/s for LU on double precision. As far as the authors know, this is the best performance measured on a heterogeneous architecture with such a large number of GPUs. Previous related works on the same problems only report results up to four GPUs using static scheduling or static data distribution. For an equivalent configuration our dynamic approach obtains similar — or better — results.

Future works include new experimental evaluations on other dense or sparse linear algebra problems such as QR factorization with new incoming accelerators such as the future Intel Xeon Phi (Intel MIC) or the next Kepler GT110 GPU.

Acknowledgment

The authors would like to thank Fabien Lementec for providing early implementations on GPU support. This work has been partially supported by the

ANR-11-BS02-013 HPAC Project, the ANR 09-COSI-011-05 Project Repdyn and CAPES/Brazil.

References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proc. of ACM SPAA. pp. 1–12. SPAA '00, ACM, New York, NY, USA (2000)
2. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Langou, J., Ltaief, H., Tomov, S.: Lu factorization for accelerator-based systems. In: Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on. pp. 217 –224 (dec 2011)
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011)
4. Ayguadé, E., Badia, R., Igual, F., Labarta, J., Mayo, R., Quintana-Ortí, E.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Proc. of Euro-Par. vol. 5704, pp. 851–862. Springer (2009)
5. Boeres, C., Chochia, G., Thanisch, P.: On the scope of applicability of the ETF algorithm. In: Proc. of the 2nd International Workshop on Parallel Algorithms for Irregularly Structured Problems. pp. 159–164. IRREGULAR '95, Springer-Verlag, London, UK, UK (1995)
6. Broquedis, F., Gautier, T., Danjean, V.: libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. In: IWOMP. pp. 102–115. Rome, Italy (2012)
7. Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguadé, E., Labarta, J.: Productive Programming of GPU Clusters with OmpSs. In: Proc. of the IEEE IPDPS (2012)
8. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing 35(1), 38–53 (2009)
9. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. pp. 212–223. PLDI '98, ACM, New York, NY, USA (1998)
10. Galilée, F., Roch, J.L., Cavalheiro, G.G.H., Doreille, M.: Athapascan-1: On-line building data flow graph in a parallel language. In: Proc. of PACT'98. pp. 88–95. IEEE Computer Society, Washington, DC, USA (1998)
11. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: Proc. of PASCO'07. ACM, London, Canada (2007)
12. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: Slaw: A scalable locality-aware adaptive work-stealing scheduler. In: Proc. of IEEE IPDPS. pp. 1 –12 (2010)
13. Hermann, E., Raffin, B., Faure, F.c., Gautier, T., Allard, J.: Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In: Proc. of Euro-Par. vol. 6272, pp. 235–246. Springer (2010)
14. Kurzak, J., Ltaief, H., Dongarra, J., Badia, R.M.: Scheduling dense linear algebra operations on multicore processors. Concurr. Comput. : Pract. Exper. 22, 15–44 (2010)

15. Lima, J.V.F., Gautier, T., Maillard, N., Danjean, V.: Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In: Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, New York, USA (2012)
16. Quintana-Ortí, G., Igual, F.D., Quintana-Ortí, E.S., van de Geijn, R.A.: Solving dense linear systems on platforms with multiple hardware accelerators. SIGPLAN Not. 44(4), 121–130 (2009)
17. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: Proc. of the IEEE IPDPS. pp. 1–8 (2008)
18. Song, F., Dongarra, J.: A scalable framework for heterogeneous GPU-based clusters. In: Proc. of ACM SPAA. pp. 91–100. ACM, New York, NY, USA (2012)
19. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing 36(5-6), 232–240 (2010)