QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators

Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, Stanimire Tomov

QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators

Emmanuel Agullo*, Cédric Augonnet*, Jack Dongarra†, Mathieu Faverge†,
Hatem Ltaief†, Samuel Thibault* and Stanimire Tomov†
*INRIA, LaBRI, University of Bordeaux, France
Email: Emmanuel.Agullo,Cedric.Augonnet,Samuel.Thibault@inria.fr
†Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996, USA
Email: Dongarra,Faverge,Ltaief,Tomov@eecs.utk.edu

***Abstract*—One of the major trends in the design of exascale architectures is the use of multicore nodes enhanced with GPU accelerators. Exploiting all resources of a hybrid accelerators-based node at their maximum potential is thus a fundamental step towards exascale computing. In this article, we present the design of a highly efficient QR factorization for such a node. Our method is in three steps. The first step consists of expressing the QR factorization as a sequence of tasks of well chosen granularity that will aim at being executed on a CPU core or a GPU. We show that we can efficiently adapt high-level algorithms from the literature that were initially designed for homogeneous multicore architectures. The second step consists of designing the kernels that implement each individual task. We use CPU kernels from previous work and present new kernels for GPUs that complement kernels already available in the MAGMA library. We show the impact on performance of these GPU kernels. In particular, we present the benefits of new hybrid CPU/GPU kernels. The last step consists of scheduling these tasks on the computational units. We present two alternative approaches, respectively based on static and dynamic scheduling. In the case of static scheduling, we exploit the a priori knowledge of the schedule to perform successive optimizations leading to very high performance. We, however, highlight the lack of portability of this approach and its limitations to relatively simple algorithms on relatively homogeneous nodes. Alternatively, by relying on an efficient runtime system, StarPU, in charge of ensuring data availability and coherency, we can schedule more complex algorithms on complex heterogeneous nodes with much higher productivity. In this latter case, we show that we can achieve high performance in a portable way thanks to a fine interaction between the application and the runtime system. We demonstrate that the obtained performance is very close to the theoretical upper bounds that we obtained using Linear Programming.**

***Keywords*-High Performance Computing; Dense Linear Algebra; QR Factorization; Hybrid Architecture; Multiple GPU Accelerators; Multicore; Tile Algorithm; Communication-Avoiding; Heterogeneous Scheduling; Runtime System.**

I. Introduction

"One algorithmic idea in numerical linear algebra is more important than all the others: QR factorization" [1]. It connects fundamental problems in the field ranging from linear systems and least squares to eigenvalue and singular value problems. It is extensively used with both dense and sparse matrices and in both direct and iterative methods for all of these problems. The QR factorization (or decomposition) of an $m \times n$ real matrix A has the form $A = QR$ where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix. If the diagonal entries of R are imposed to be positive, this decomposition is unique when A is non singular. Different structures for A may arise depending on the applications. The most important structural property is whether the matrix is sparse or dense. The shape of the matrix is usually non square ($m \neq n$). For instance, the first, predominant step of the standard algorithm for solving dense least square problems (as implemented in LAPACK) is the QR factorization of a dense matrix A representing an overdetermined system ($m > n$). When applied to a square matrix ($m = n$), the same algorithms become a stable method to solve the corresponding linear system. Although the LU decomposition is usually preferred in this latter case since its cost is twice lower, square matrices are conventionally used to benchmark the performance of a QR factorization. The performance of the QR factorization of a matrix such that $m \gg n$, so called *tall and skinny*, is also very critical. For instance, it is fundamental for solving large sparse linear systems with Krylov space iterative methods such as the Generalized Minimal Residual Method (GMRES), since it is often used in practice to perform a subspace minimization step of each iteration.

Several algorithms can perform a QR decomposition. A simple route to understanding this decomposition is to view it as a direct application of the well known Gram-Schmidt process, consisting of orthonormalizing a set of vectors. The application of this process to the column vectors of a matrix A leads to its QR decomposition. However, in the presence of rounding errors on a computer, this process, known also as Classical Gram-Schmidt (CGS), turns out to be unstable in the sense that Q quickly looses its orthogonality. A more stable variant of this algorithm, the so called Modified Gram-Schmidt (MGS), ensures that the computed QR factors have a small residual. However, the computed Q is guaranteed to be nearly orthonormal only when the matrix is well conditioned [2]. Algorithms based on unitary transformations (Givens rotations or Householder reflections) are stronger because they unconditionally ensure this property. Givens rotations may be privileged for zeroing out a few elements of a matrix whereas Householder trans-

formations are more efficient for zeroing out a full vector of a matrix. Therefore, the QR factorization of dense matrices in reference software has consistently been implemented as a succession of elementary Householder transformations of the form $H = I - \tau v v^T$ where I is the identity matrix, v is a column reflector and τ is a scaling factor [3]. This algorithm, implemented in LINPACK in the 1970s, has been redesigned at the pace of the successive revolutions of hardware in supercomputers. LAPACK took advantage of hierarchical levels of memory thanks to a two-step process that formed the basis for the so called *blocked* algorithms [4]. A limited set of columns is factored (*panel factorization*) using Householder transformations as in LINPACK. The transformations used are accumulated [4] and applied all at once to the trailing submatrix (*update step*). The process is repeated until all columns have been factored. SCALAPACK furthermore took advantage of distributed memory machines by parallelizing both the panel factorization and the update step.

Although the design of exascale architectures is still an intensive research and development topic, next generation supercomputers are likely to be machines composed of thousands of accelerators-based multicore nodes. This new hardware paradigm raises multiple challenges that the algorithm and software communities have to face. Exploiting the potential of these architectures at their maximum performance is one of the key challenges. Two complementary research directions may be investigated. One is the scalability of the algorithms on a large number of nodes [5], since the number of cores of top supercomputers are growing at an unprecedented scale. The other one consists of designing highly optimized algorithms that fully exploit all the resources available on a single node. In this paper, we discuss this second aspect by presenting a new method for performing the QR factorization of dense matrices on a multicore node enhanced with multiple GPU accelerators.

Our method consists of three steps. The first step involves expressing the QR factorization as a sequence of tasks of well chosen granularity that will aim at being executed on a CPU core or a GPU. Such a sequence can be viewed as a high-level algorithm. We chose two high-level algorithms from recent literature that were initially designed for homogeneous multicore architectures. We recall these algorithms, Tile QR and Tile Communication-Avoiding QR (Tile CAQR), and explain this choice in Section II. The second step consists of designing the kernels that implement each individual task. We use CPU kernels from previous work [6] and design new kernels for GPUs (Section III). Some of our GPU kernels are actually *hybrid* since they are executed on a GPU and associated to a CPU core. We exploit their design with algorithmic considerations and validate the choices thanks to a performance study. The last step consists of scheduling these tasks on the computational units. We present two alternative approaches respectively based on static and dynamic scheduling. In the case of static scheduling (Section IV), we exploit the a priori knowledge of the schedule to perform successive optimizations leading to very high performance. We, however, highlight the lack of portability of this approach and its limitations to regular algorithms on relatively homogeneous nodes. Alternatively, by relying on an efficient runtime system, STARPU [7], in charge of ensuring data availability and coherency, we can benefit from all the resources available on a complex heterogeneous node (Section V) and schedule more advanced algorithms (Section VI) with much higher productivity. We show that this latter approach combines very high performance and portability thanks to a fine interaction between the application and the runtime system. In particular, we demonstrate that the obtained performance is very close to theoretical upper bounds that STARPU can automatically exhibit using Linear Programming (LP).

The rest of the paper is organized as follows. In Section II, we present the Tile QR and Tile CAQR high-level algorithms that we chose from the literature. We also present there the STARPU runtime system, related work, and the experimental environment. The design of kernels to be executed on the devices (CPU cores or GPUs) is discussed in Section III. Sections IV and V discuss methods for executing the high-level Tile QR algorithm using static and dynamic scheduling, respectively. Section VI shows the benefits of using the more advanced Tile CAQR algorithm when processing tall and skinny matrices.

II. BACKGROUND

A. Tile QR and Tile CAQR factorizations

LAPACK and SCALAPACK are the current de facto standard libraries for performing advanced linear algebra operations. Their QR factorization is designed as a high-level algorithm relying on more basic routines (such as a matrix multiplication) from the Basic Linear Algebra Subprograms (BLAS) and Parallel BLAS (PBLAS) packages, respectively. Both libraries may perform parallel executions. On one hand, LAPACK was initially designed for sequential machines but may benefit from parallel BLAS implementations for shared memory machines, such as vendor BLAS. On the other hand, SCALAPACK was specifically designed to scale on distributed memory machines. As discussed in Section I, the QR factorization in LAPACK and SCALAPACK is based on the same high-level algorithm that performs blocking in order to take advantage of hierarchical memories. The blocking strategy consists of performing a panel factorization followed by an update step until all the panels are factored. In the reference implementation of LAPACK, a synchronization implicitly occurs after each BLAS call (occurring at each update step). This fork-join approach only achieves an extremely low ratio of the theoretical peak of recent multicore architectures [8]. In SCALAPACK, synchronizations are alleviated thanks to a

two-dimensional cyclic mapping of the processes (which also enables efficient load balancing). Superior to the fork-join LAPACK reference model, this approach still achieves a limited fraction of the peak [8]. Higher performance may be achieved with this panel-update approach by performing a look-ahead technique. Look-ahead consists of splitting the update step into two parts. The update of the next panel is performed before the rest of the trailing submatrix in order to interleave non-efficient BLAS-2 panel factorizations with the efficient BLAS-3 updates. Along with multiple unpublished optimizations, vendor libraries (e.g., Intel MKL or IBM ESSL) provide LAPACK implementations enhanced with look-ahead. This approach allows to hide the cost of the slow panel factorization when the matrices are large enough and when a limited number of cores is used. In such conditions, these libraries can reach a high proportion of the theoretical peak of multicore machines. However, when the number of cores is large compared to the matrix size, the cost of panel factorizations is dominant and their performance is limited [8].

So called *tile algorithms* enable us to break the panel factorization into tasks of fine granularity. This approach takes its roots in *updating factorizations* [3]. Using updating techniques to obtain tasks of fine granularity was first exploited in the context of out-of-memory (often called *out-of-core*) factorizations [9]. Two implementations of the Tile QR factorization for multicore architectures were more recently proposed [10], [11]. Tile QR annihilates matrix elements by tiles (square blocks) instead of rectangular panels as in LAPACK. In this paper, we use the algorithm presented in [10], which relies on the following four kernels:

`geqrt:` this routine performs the QR factorization of a diagonal tile A_{kk} of size $nb \times nb$ of the input matrix. It produces an upper triangular matrix R_{kk} and a unit lower triangular matrix V_{kk} containing the Householder reflectors. An upper triangular matrix T_{kk} is also computed as defined by the WY technique [4] for accumulating the transformations. R_{kk} and V_{kk} are written on the memory area used for A_{kk} while an extra work space is needed to store the structure T_{kk}. The upper triangular matrix R_{kk}, called *reference tile*, is eventually used to annihilate the subsequent tiles located below, on the same panel.

`tsqrt:` this routine performs the QR factorization of a matrix built by coupling the reference tile R_{kk} that is produced by `geqrt` with a tile below the diagonal A_{ik}. It produces an updated R_{kk} factor, a matrix V_{ik} containing the Householder reflectors and a matrix T_{ik} resulting from accumulating the reflectors V_{ik}.

`ormqr:` this routine applies the transformations computed by `geqrt` to a tile A_{kj} located on the right side of the diagonal tile.

`tsmqr:` this routine applies reflectors V_{ik} and matrix T_{ik} computed by `tsqrt` to tiles A_{kj} and A_{ij}.

Since the Tile QR factorization is also based on Householder reflectors that are orthogonal transformations, this factorization is stable. Figure 1 shows the first panel reduction applied on a 3×3 tile matrix. The triangular shapes located on the left side of the matrices correspond to the extra data structure needed to store the different T_{ij} triangular matrices. The gray tiles represent the input dependencies for the trailing submatrix updates. As expected, the panel factorization is broken into three operations (left column in Figure 1) triggering the updates (central and right columns) before the panel is fully factored. Such a fine granularity allows the algorithm to achieve better performance on recent multicore machines [8]. However, the panel factorization (left column) has still to be performed in sequence (flat tree, in the terminology of [12]). When the matrix has few panels (such as tall and skinny matrices discussed, see Section VI), the approach does not exhibit enough parallelism to fully exploit all the devices of modern architectures.

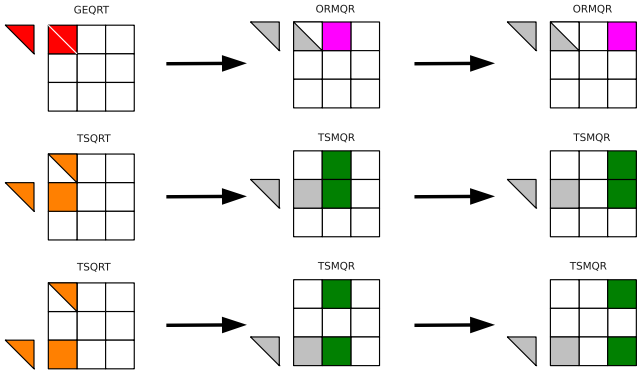

Figure 1. Tile QR algorithm: Factorization of the first column of tiles (left) and corresponding updates (center and right).

The "Communication Avoiding" QR [12] (CAQR) algorithm introduces parallelism in the panel factorization itself [12]. The matrix is split into *domains* (or block rows) that are processed concurrently within the factorization of a panel and then merged according to a binary tree. Tile CAQR consists of combining Tile QR and CAQR. In the terminology of [12], Tile CAQR consists of hybrid binary/flat tree. Tile CAQR was recently proposed in the context of multicore architectures [6]. The principle is as follows. Each domain is internally factored with the Tile QR algorithm (flat tree) and the reduction of the domains is performed in parallel (binary tree). Tile CAQR needs two extra kernels for merging the domains according to a binary tree:

`ttqrt:` this routine performs the QR factorization of a matrix built by coupling both upper triangular reference tiles $R_{p1,kk}$ and $R_{p2,1k}$ produced by the `geqrt` of two domains. It produces an updated factor $R_{p1,kk}$, a triangular matrix $V_{p2,1k}$ containing the Householder reflectors and a matrix $T_{p2,1k}$. resulting from accumulating the reflectors $V_{p2,1k}$.

`ttmqr:` this routine applies reflectors $V_{p2,1k}$ and matrix $T_{p2,1k}$ computed by `tsqrt` to $A_{p1,kj}$ and $A_{p2,1j}$.

Both Tile QR and Tile CAQR we refer to in the present paper are fully described in [6]. Compared to the CAQR algorithm initially implemented in [12] (binary tree), our hybrid tree adapts the algorithm to the effective platform topology by choosing the number of domains. For instance, for large square matrices, we would use one domain (which corresponds to the standard Tile QR algorithm) whereas the number of domains may be augmented when the number of rows increases (for a fixed number of columns). While Tile QR is regular enough to achieve an efficient load balancing with a static schedule on homogeneous platforms (Section IV), Tile CAQR is much more irregular and needs more advanced runtimes (Section VI) to be efficiently scheduled.

B. Related work

Various projects provide GPU kernels for linear algebra. The CULA library implements many BLAS and LAPACK kernels [13], but those are limited to a single GPU and problems that fit into the memory. Kurzak *et al.* accelerated the QR decomposition of the PLASMA library with a single GPU by designing non standard BLAS kernels. Anderson and Demmel also provide a highly optimized hand-coded CAQR single-GPU kernel written directly in CUDA [14]. The PLAPACK [15] and the DPLASMA [16] linear algebra libraries both target accelerator-based clusters. Both of them rely on their own runtime systems that provide intra-node dynamic scheduling capabilities.

Different runtime systems were designed to support accelerator-based platforms. The StarSs [17] language is an annotation-based language that can either be executed on CPUs, GPUs or Cell processors, respectively, using SMPSs, GPUSs or CellSs. Diamos and Yalamanchili also propose to use features that are similar to those available in StarPU in the Harmony runtime system [18]. Sequoia [19] statically maps hierarchical applications on top of clusters of hybrid machines, and the Charm++ runtime system was extended to support GPUs [20].

C. Experimental environment

All the proposed algorithms are evaluated on two hybrid systems, which we name "Nehalem-Quadro" (based on Intel Nehalem CPU and NVIDIA Quadro GPU) and "Opteron-Tesla" (based on AMD Opteron CPU and NVIDIA Tesla GPU). Their characteristics are summarized in Table I. We detail how to read the table for the first platform. This Nehalem-Quadro platform is composed of two quad-core Intel Nehalem X5550 CPUs (8 CPU cores total) running at 2.67 GHz with 48 GB of memory divided in two Non Uniform Memory Access (NUMA) nodes. It is enhanced with three NVIDIA Quadro FX5800 GPUs of 240 cores each (720 GPU cores total) running at 1.3 GHz with 4 GB of GDDR3 per GPU. We use CUDA 3.2. In single precision (SP), the peak performance of a CPU core is 21.3 Gflop/s (170.4 Gflop/s for all 8 CPU cores) and 622 Gflop/s for a GPU (1866 Gflop/s for all three GPUs). The SP theoretical peak of the machine is thus equal to 2036 Gflop/s. The SP matrix multiplication (`sgemm` from GOTOBLAS2 library) peak is about 20.6 Gflop/s on a CPU core and 343 Gflop/s on a GPU. This sums up to a *cumulated sgemm peak* (that we note $\sum$`sgemm`) equal to 1200 Gflop/s. Since `sgemm` is faster than all our kernels, this value is an upper bound on the SP performance we may obtain. In double precision (DP), the cumulated CPU and GPU peaks are equal to 85.4 Gflop/s and 234 Gflop/s, respectively, for a total machine peak of 319.4 Gflop/s. The DP cumulated `dgemm` peak ($\sum$`dgemm`) is equal to 10.3 Gflop/s per CPU and 73.5 Gflop/s per GPU, for a total machine peak of 302.9 Gflop/s.

III. HYBRID CPU/GPU KERNELS FOR TILE QR AND TILE CAQR FACTORIZATIONS

The development of highly optimized kernels is crucial for the overall performance of the hybrid algorithms proposed. The tile QR and tile CAQR factorizations can be naturally described using correspondingly four and six main kernels. For each of these kernels an implementation has to be provided for at least one of the hardware components available. This enables a runtime system to then take a high-level algorithm description and to schedule (statically or dynamically) the tasks execution over the heterogeneous hardware components at hand. The following three sections specify the main aspects of our approach as related to tuning the tasks' granularity, developing multicore CPU kernels, as well as GPU CUDA kernels.

A. Kernels for (single) CPU

We distinguish two general ways to code dense linear algebra for a particular hardware component: BLAS based and fused BLAS. The benefit of using BLAS based is that optimized BLAS implementations can be leveraged immediately. This is the LAPACK approach, for example, and in general a preferred one for CPU kernels development. We use CPU kernels that were already developed as part of the PLASMA library (plus two more kernels from previous work [6] for performing the merge operations in the case of Tile CAQR). The kernels are based on sequential optimized BLAS. Speeding up these kernels by fusing certain BLAS requires the availability of optimized BLAS sources. This is often challenging to obtain, understand, and/or modify, as it is often written in assembly, making it also not portable. Attempts to speedup the QR kernels by fusing certain BLAS could not overcome these challenges.

B. GPU CUDA kernels

Similar to the CPUs' kernels development, writing code without relying on BLAS, e.g., entirely in CUDA, is challenging. On the other hand, having all BLAS is often not feasible either. For example, the panel factorizations are composed of small BLAS calls that are memory bound

Table I
"NEHALEM-QUADRO" AND "OPTERON-TESLA" MACHINES (NAMED AFTER THEIR CPU-GPU ARCHITECTURE)

Machine	#procs. (#cores)	Freq. (GHz)	Memory (GB)	SP peak (Gflop/s)	$\sum$sgemm (Gflop/s)	DP peak (Gflop/s)	$\sum$dgemm (Gflop/s)
Intel Nehalem X5550 CPU	2 (8)	2.67	48 (2 × 24)	170.4	165	85.4	82.4
NVIDIA Quadro FX5800 GPU	3 (720)	1.3	12 (3 × 4)	1866	1035	234	220.5
Nehalem-Quadro (sums)				2036	**1200**	319.4	**302.9**
AMD Opteron 8358 SE CPU	4 (16)	2.4	32 (4 × 8)	307.2	256	153.6	131.2
NVIDIA Tesla S1070 GPU	4 (960)	1.3	16 (4 × 4)	2760	1652	345	336
Opteron-Tesla (sums)				3067.2	**1908**	498.6	**467.2**

and often do not have enough parallelism to be efficiently executed on a GPU. It has been discovered that it is better to offload them to a CPU, leading to the development of a number of hybrid algorithms for one and two-sided factorizations [21], [22]. These algorithms are available in the MAGMA library and we use some of them in the QR and CAQR algorithms. For example these are the kernels for QR factorization on a tile (`geqrt`) and the application of Q on the trailing tiles (`ormqr`). Two new kernels that had to be developed are described bellow. The kernels are based on BLAS. One has a hybrid component and the other is fusing BLAS kernels. The fusing contributed to higher performance as we have the sources for the current state-of-art CUDA BLAS kernels [23]. The alternative – to develop everything in CUDA [14] – can lead to high performance but has its development challenges in terms of using fastest BLAS available and maintenance, as already pointed out.

A hybrid `tsqrt` kernel: This kernel is developed similarly to the hybrid QR algorithm in MAGMA– panels are sent and factored on the CPU, and trailing matrix updates are done on the GPU. Overlap is achieved between the work on the CPU and the GPU through lookahead techniques. The panel factorization on the CPU uses the corresponding kernel from the PLASMA library. For panel j the orthogonal Q_j factors have the form

$$Q_j = I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T,$$

where V_j are the reflectors and T_j is upper triangular. Q_j is applied from left to the trailing matrix on the GPU using fused `gemm` operations (only). The application is the main component of the `tsmqr` kernel and is explained next.

A CUDA `tsmqr` kernel: This is the most time consuming kernel. It applies a sequence of Q_j transformations to pairs of corresponding trailing tiles, e.g.,

$$\begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix} = \left[I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T \right] \begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix},$$

where A^j_{ki} is a block of rows of tile A_{ki}. Namely, if we denote the inner blocking size by ib, these would be rows from $j\ ib$ to $(j+1)\ ib$. We have implemented this with the following three CUDA kernels properly modifying state-of-the-art CUDA `gemm` sources:

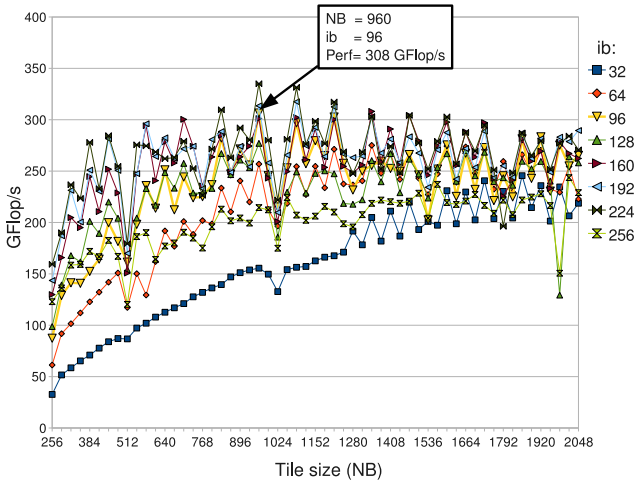

Figure 2. Tuning the performance of the `stsmqr` kernel on a GTX280 GPU. Tile size of 960 and inner-blocking of 96 is selected on this GPU.

(1) $D^1_{work} = A^j_{ki} + V_j^T A_{mi}$;
(2) $D^2_{work} = T_j D^1_{work}$; $A^j_{ki} = A^j_{ki} - D^2_{work}$;
(3) $A_{mi} = A_{mi} - V_j D^2_{work}$.

For the CAQR algorithm we provide two GPU kernels. These are a hybrid QR factorization of panels (the hybrid `geqrt` kernel from MAGMA applied to tall and skinny panels) and the application of the resulting Q to the trailing tiles (most time consuming; this is `ormqr` from MAGMA). The other four kernels are of very small granularity and are left for execution entirely on the multicore.

C. Granularity selection

High level algorithms may often be parametrizable by the granularity of the task. The granularity selection provides a trade-off between overall parallelism available (with "small" granularity) and kernel performance (with "large" granularity). Finding "optimal" granularity values that would maximize the overall performance is thus important and challenging. As hardware evolves in complexity, it becomes more difficult to predict performance, e.g., based on a model. Empirical tuning approaches, based on feedback from experiments, become increasingly popular in the area of developing and tuning high-performance linear algebra libraries [24], [25]. In Tile QR and Tile CAQR algorithms,

the granularity of the task is the tile size. Our kernels' implementations are parametrizable to facilitate ease of tuning. However, because the search space for tuning the whole algorithm becomes quickly very large, we perform the tuning to the kernels' level. The advantage is that a kernel operates on tiles (of order 1000). It it thus faster to perform the tuning doing so than if we had to tune the high level algorithm that operates on a size of order equal to several thousands. On the other hand, our approach is not as accurate, since it does not capture the whole algorithm that we want to optimize. A good approximation consists of choosing the tile size based on the most compute intensive kernel. For large square matrices, `sgeqrt` dominates the computation. Therefore we choose our tile size based on its behavior.

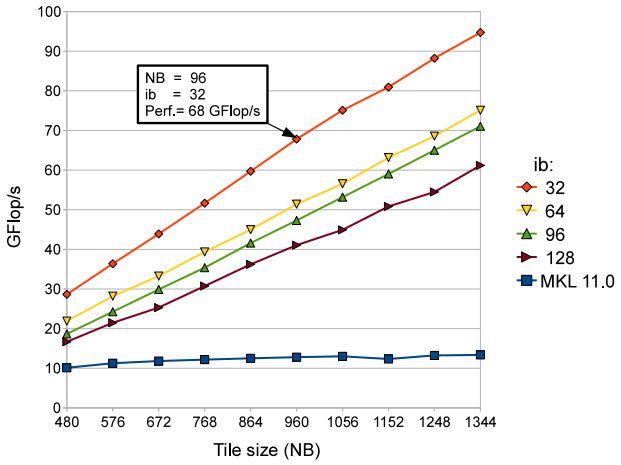

Figure 3. Tuning the performance of the `sgeqrt` kernel on a GTX280 GPU. Tile size of 960 is selected based on tuning the `stsmqr` kernel.

The kernel is run for multiple tile sizes (NB) and inner-blocking sizes (ib) (Figure 2). Several feasible combinations are chosen, the entire algorithm is run only on those choices, and based on the performance obtained, we fix the NB and ib. Note that although in this case the finally selected NB/ib give us the best overall performance, this selection does not give best performance for the `stsmqr` kernel (which is usually the case). The performance numbers are for the GTX280 GPU. The same experiments and tuning are carried out for each GPU of interest.

IV. Static Scheduling

This section describes the mechanism of the static runtime system used as a first attempt to schedule the previously defined kernels over the hybrid components.

A. Basic Concepts

Originally implemented to schedule the Cholesky and QR factorizations on the Cell processor [26], the hand-coded static scheduler dispatches the hybrid kernels across all CPU-GPU pairs available on the system in one dimensional cyclic fashion. This runtime imposes a linear scheduling order on all the kernels during the factorization. This order enforces the execution of a predetermined subset of kernels on a particular CPU-GPU pair.

There are two global progress tables to ensure numerical correctness and to get high performance, respectively. The first progress table keeps track of dependencies among the different tasks at different steps of the factorization. A dependency check is performed before executing each kernel by examining the local copy of the progress table. The hosting CPUs stall with busy waiting on volatile variables until the corresponding dependencies are satisfied, which simultaneously triggers the release of the kernel to the designated CPU-GPU pair. The second progress table considerably decreases the number of communication involved between a CPU-GPU pair, which is critical given that the PCI bus is two orders of magnitude less efficient than the computational power of the accelerators. This latter table emulates, to some extent, the cache coherency protocol to synchronize data present in the host and the device memories, whenever necessary.

B. Looking Variants

The right-looking variant (RL) is actually the default static scheduler version of the tile QR factorization on homogeneous multicore systems. Basically, this variant consists in updating first the trailing submatrix, located on the right side of the current panel, before the execution of the next panel starts. This looking variant generates many tasks which can potentially run in parallel.

The left-looking variant (LL), also called the *lazy* variant, applies all subsequent updates generated from the left side to the current panel before proceeding with the next panel. Since the updates are restricted to the panel only, data reuse is maximized while at the same time parallelism gets limited.

The methodology of this static scheduling allows for pipelined execution of factorizations steps, which usually provides similar benefits to dynamic scheduling to some extent. For example, the execution of the inefficient Level 2 BLAS operations can be overlapped by the efficient Level 3 BLAS operations. This phenomenon has been successfully demonstrated for the scheduling of one-sided factorizations in the context of homogeneous multicore [27]. However, when tackling hybrid system components, i.e., multicore associated with GPU accelerators, a tremendous gap in terms of sequential performance may exist between the hybrid CPU-GPU kernels and the GPU kernels as seen in Section III. Therefore, one of the main disadvantages of the static scheduling is a potential suboptimal scheduling, i.e., stalling in situations where work is available.

Figure 4 clearly describes this drawback by depicting three traces of the tile hybrid QR on four CPU-GPU pairs. The dark purple colors represent the panel tasks and the

light green colors are the update kernels. The panel tasks are hybrid and the GPU needs the CPU to perform the level 2 BLAS operations while the update kernels are highly efficient level 3 BLAS operations performed on the GPU only. The top graph shows the RL variant with lots of stalls. The panel tasks indeed become a bottleneck and the updates tasks cannot proceed until the completeness of the panel tasks. The middle graph presents the LL variant. The scheduling contains less gaps but still suffers from the lack of parallelism, especially in the beginning. And this inefficiency is even more exacerbated by the slow panel hybrid kernels.

A new looking variant has then been implemented to alleviate this bottleneck combining the previous LL version with a breadth-first search task execution (BF-LL). Each CPU-GPU pair applies all subsequent transformations, once for all, (update and panel tasks) on a particular tile on the current panel before proceeding with the tile below it. The obtained trace is very compact and dense as shown in the bottom trace. Noteworthy to mention is that some load imbalance starts to appear toward the end.

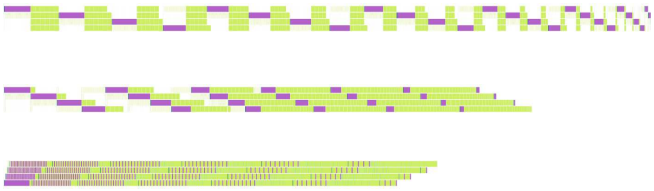

Figure 4. a) Top: RL b) Middle: LL c) Bottom: BF-LL

C. Preliminary Results

Figure 5 shows performance numbers of the different looking variant in single and double precisions, respectively, on Opteron-Tesla machine using four CPU-GPU pairs. We take for this experiment a block size nb of 960 and an inner-blocking size ib of 96 and 64 for single and double precisions, respectively. These sizes have been selected after tuning, as presented in the section III-C.

As pointed out by the traces of the figure 4, the "BF-LL" provides better performances than the two other variants. It achieves more than 50% of the gemm peak of the GPUs cumulated compare to the RL variant, which achieves only 33% of the gemm peak in single.

Figure 6 presents the scalability of the "BF-LL" variant in single and double precision, respectively. The performance clearly doubles with the number of CPU-GPU pairs.

D. Critical Limitations

There are basically three main limitations of using this static scheduler and alike in a heterogeneous environment. First, getting the appropriate static scheduling variant in order to finally generate decent performance numbers is a very challenging and time-consuming exercise, especially when

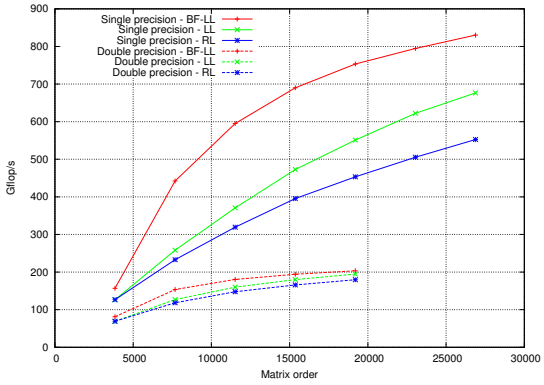

Figure 5. Impact of the looking variant on Opteron-Tesla.

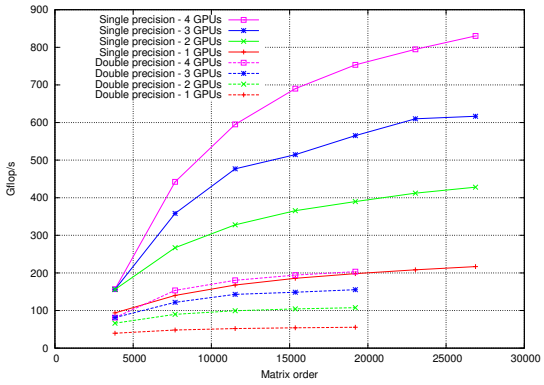

Figure 6. Scalability of the BF-LL variant on Opteron-Tesla.

dealing with heterogeneous systems. Moreover, generally speaking, a static scheduler cannot efficiently accommodate dynamic operations, e.g., *divide-and-conquer* algorithms or even a more complex algorithm like Communication-Avoiding QR (see Section VI). So, the productivity becomes a natural concern. Second, the scheduling is restrained only to CPU-GPU pairs, and the available resources provided by the homogeneous multicore remain idle. Last but not least, the size of the matrix to be factored is limited because of the necessary large amount of memory allocated on the different GPUs to prevent communication overheads.

V. Dynamic scheduling with StarPU

StarPU is a runtime system that schedules tasks on CPU cores and GPUs. A task is a function working on data and those two notions are thus central to StarPU. First, the application has to register all data into StarPU. It then does not access it anymore through its memory address but through a StarPU abstraction, the *handle*, returned by registration. StarPU transparently guarantees that a task that needs to access a piece of data will be given a pointer to a valid data replicate. It will take care of the data movements and therefore relieve programmers from the burden of explicit data transfers. Second, a multi-version

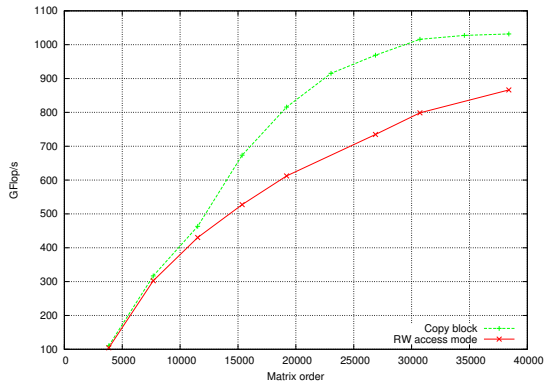

Figure 7. Copying the diagonal block to save parallelism

kernel, the *codelet*, is a gathering of the kernel implementations available for the different devices (CPU core and GPUs). In the end, a task can be defined independently of the device as a codelet working on handles and submitted to StarPU. The tasks are then executed according to a scheduling optimization strategy.

A. Tile QR on top of StarPU

In this implementation of the Tile QR algorithm, the CPU kernels are those from the PLASMA library presented in section II-A, and the GPU kernels are based on the MAGMA hybrid kernels that were modified to produce exactly the same output as the PLASMA kernels. All the tasks that compose the DAG of the Tile QR algorithm are submitted asynchronously to StarPU, which automatically infers task dependencies based on data dependencies. In essence, porting the Tile QR algorithm from the dynamically scheduled PLASMA library on an accelerator-based machine thus essentially requires providing the CUDA kernels from MAGMA. However, a naive implementation achieves low performance. We discuss in the next section how we successively improved the performance up to almost reaching the upper theoretical bound that we exhibited.

B. Adapting the kernels and the algorithm for dynamic scheduling on an hybrid platform

We had to modify MAGMA hybrid kernels so that they reconstruct exactly the same output as their CPU counterparts. This overhead is nevertheless balanced by the fact that we provide more flexibility to the scheduler since we do not need to execute all the tasks related to a common panel factorization on the same GPU and/or CPU.

Data coherency is maintained at the tile level because transferring non-contiguous pieces of data with a non-constant stride such as triangles is highly inefficient. However, the `tsqrt` kernels modify the upper triangular of the diagonal block while the `ormqr` kernels concurrently access the lower part of the same tile in a read-only fashion (see figure 1). Contrary to the shared-memory implementation

Table II
SCHEDULING STRATEGIES IMPLEMENTED WITH STARPU

Name	Policy description
greedy	Greedy policy
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

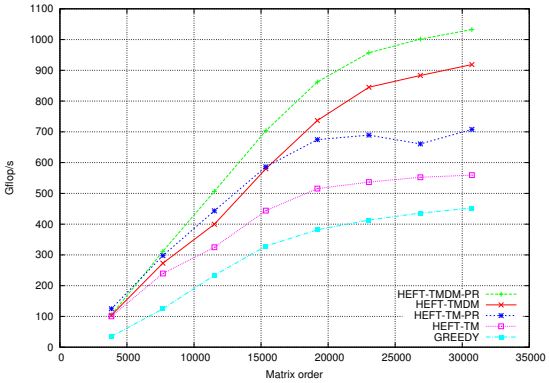

Figure 8. Impact of the scheduling policy on Opteron-Tesla

of the Tile QR algorithm found in PLASMA, we cannot avoid serializing all these kernels by pretending that there is no dependencies between these tasks. Instead, we act as if the `ormqr` and `tsqrt` kernels were accessing the entire diagonal block, respectively, in a read-write and a read-only mode creating useless dependencies. The solution is to create a copy of this diagonal block so that the `ormqr` kernel can access the upper part of the block in read-only mode. This solution avoids serializing the `ormqr` kernels, and therefore provides more parallelism. It also reduces false-sharing cache effects that may affect the performance of kernels accessing concurrently to the same tile. Integrating such a copy of the diagonal in the existing algorithm is straightforward with StarPU, as we just have to add a task that copies the diagonal block into an extra tile. Figure 7 shows the performance obtained either by always accessing the diagonal block in a read-write mode or by having the `ormqr` access a replica of that block. The drawback of this approach is a little extra memory consumption, which corresponds to the size of a panel ($nb \times n$, where nb is the size of a block and n the number of rows in the matrix).

Table III
IMPACT OF THE SCHEDULING POLICY ON THE TOTAL AMOUNT OF DATA TRANSFERS DURING SGEQRF ON NEHALEM-QUADRO

Matrix order	9600	24960	30720	34560
heft-tm-pr	3.8 GB	57.2 GB	105.6 GB	154.7 GB
heft-tmdm-pr	1.9 GB	16.3 GB	25.4 GB	41.6 GB

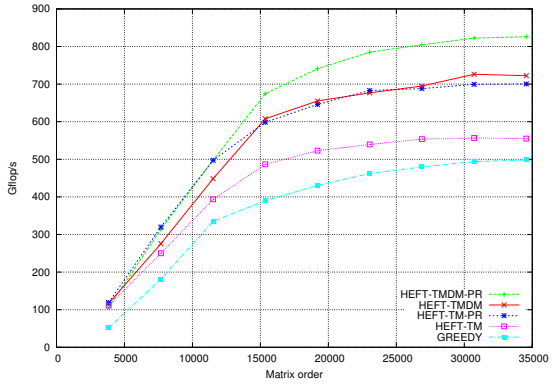

Figure 9. Impact of the scheduling policy on Nehalem-Quadro

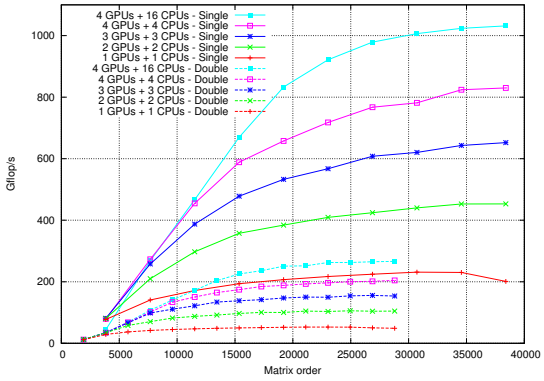

Figure 10. Scalability of `SGEQRF` and `DGEQRF` on Opteron-Tesla

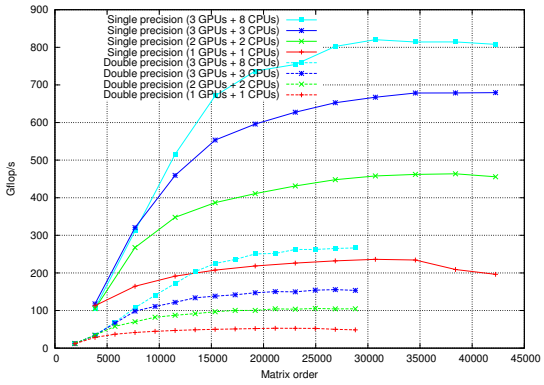

Figure 11. Scalability of `SGEQRF` and `DGEQRF` on Nehalem-Quadro

C. Impact of the scheduling policy

A high-level interface is available to design customized policies tailored for the specific needs of the application, and StarPU provides various built-in scheduling policies. Figures 8 and 9 shows the performance obtained by the different scheduling policies on both machines with different numbers of GPUs and CPUs.

The greedy policy corresponds to a single centralized queue. While it provides work to any idle processing unit as long as there remains tasks, this strategy does not take heterogeneity into account and may assign a critical task to a slow processing unit.

We therefore implemented the HEFT scheduling strategy [28] that minimizes the termination time of the different tasks. This relies on StarPU's auto-tuning mechanisms which transparently predict the length of the tasks. The so-called HEFT-TM strategy takes heterogeneity into account and gives better load balancing results [29].

In the context of multi-accelerator platforms, the bus is a scarse resource so that it is crucial to hide the overhead of data transfers, or even to avoid transferring data whenever possible. We extended the HEFT-TM policy with a *prefetch* mechanism; when a task is assigned to a processing unit, StarPU starts transferring all its data in advance (HEFT-TM-PR). On both our multi-GPU platforms, hiding transfer overhead with the prefetch mechanism indeed yields a significant speed improvement.

However, this does not reduce the amount of transfers, and therefore the pressure on the bus. When minimizing the cost required to execute a task on a processing unit, we can not only consider the execution time, but also the time required to transfer input data (which StarPU is able to predict as well). We therefore modified the HEFT-TM policy to take both the task duration and data transfers into account when minimizing the execution time of a task (HEFT-TMDP). As shown in Table III, this technique drastically reduces the total amount of data transfer because the scheduler can detect that it is more expensive to move data than to execute a task locally. The impact of these optimizations is getting even more important as the number of accelerators increases. Finally, we combined both HEFT-TMDP and HEFT-TM-PR strategies to avoid data transfers and to hide the overhead of the transfers that are really required (HEFT-TMDP-PR).

All other measurements of this paper use this HEFT-TMDP-PR strategy, which provides good balancing, reduces the amount of data transfers and hides their overhead. More details on HEFT-TMDP-PR are available in our previous work [29].

D. Performance portability

Figures 10 and 11 show the performance obtained using either using only the GPUs or using all CPUs in addition to the GPUs. On both machines, we obtain almost a perfect speedup with respect to the number of GPU-CPU pairs. It is worth noting that the speed is similar to that of the static scheduling previously shown in Section IV. Contrary to the static approach, we are not limited to problems that fit into the 4 GB of memory on each of the GPUs; we even obtain super-linear speedup when the problem is too big to fit into a single GPU. StarPU also permits using all the CPUs in addition to the GPUs: adding 12 CPUs on Figure 10 (resp. 5 CPUs on Figure 11) gives an improvement of 200 Gflop/s

(resp. 130). This is especially interesting because the peak performance is about 10 Gflop/s (resp. 20) per CPU core; this significantly exceeds the peak performance of 12 cores (resp. 5). This super-linear acceleration that we have already observed to a lesser extent in our previous work [7] is explained by our heterogeneity-aware scheduling policies. The rationale is that StarPU knows that some kernels are not so efficient on GPUs, and thus schedules them on CPUs to save GPU power for other tasks. We thus have super-linear speedups because the processing units are doing only what they are good for.

Figure 11 also shows the performance of the Tile QR algorithm in double precision. Even though the GPUs perform about 8 times slower on double precision than on single precision while this factor is only 2 on the CPUs, we still obtain portable performance on the different platforms. In that case, the benefit resulting from the use of extra CPUs is relatively more important than in single precision.

E. Comparison with theoretical upper bounds

The successive scheduling optimizations described in this paper provide a lot of various performance improvements. It is thus questionable whether the obtained results can still be improved a lot. We now discuss a few theoretical upper bounds that we compare to our best execution case on the Nehalem-Quadro machine (shown on Figure 12).

The theoretical peak of the machine and the accumulated `sgemm` performance presented in Section II-C are obvious upper bounds, the latter being represented by the first curve of the figure. They are however very coarse, as they do not take into account that the QR factorization is composed of several kernels with varying general efficiency as well as varying GPU/CPU relative performance. To properly and easily estimate that, we have added to StarPU an optional mode that records the number of executed tasks according to their type (the kernel they run and their operand size).This can then be combined with the respective performances of the kernels, which StarPU can provide thanks to its history-based performance models [29]; estimating an upper bound for the execution time resorts to solving the corresponding LP problem. StarPU was extended to optionally compute this bound so it can be easily printed along with other timing information. This is performed with relaxation as the difference with integer resolution is negligible for non-tiny sizes. This provides the second curve of the figure, which provides a better upper bound since it is optimal according to the heterogeneity of both task types and workers.

To get an even more precise upper bound, we need to take task dependencies into account. We thus use a Mixed-Integer Linear Programming (MILP) problem, in which we distinguish all tasks independently, and can then introduce dependency constraints. We have extended StarPU to optionally emit such a problem automatically from the actual execution

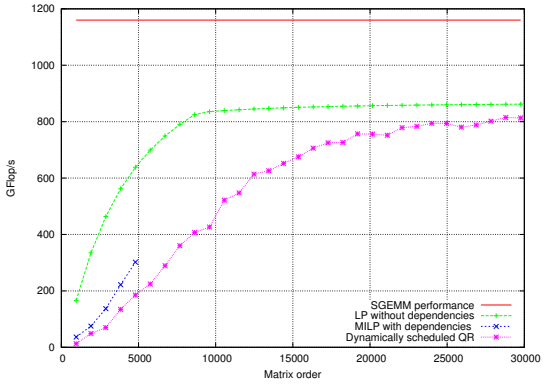

Figure 12. Dynamic scheduling results *vs* theoretical boundaries on Nehalem-Quadro

of any application. For a 5x5-blocked QR factorization, *i.e.* 60 tasks, this typically results in about 1400 variables (of which 1250 are binaries) and 13000 constraints (most of which are Mixed-Integer). This is the biggest size for which the resolution is possible with reasonable time. The third curve shows that thanks to taking dependencies into account, this provides a much better upper bound, as dependencies are what typically reduce parallelism and possibilities for optimizations. Even if few values are available, this already provides a good indication that the performance obtained by our scheduling optimizations is already very close to the optimum that could be achieved. This bound could be further refined by also modeling data transfers, but this would produce yet more variables and constraints while most transfers are actually overlapped with computation.

F. Advantages and limitations of dynamic scheduling

A dynamic scheduler cannot make decisions based on tasks that have not been submitted yet. This typically results in slightly lower performance for small problems because any under-optimal decision may directly affect the overall performance. It is also harder to implement the performance oriented optimizations that are typically found in HPC libraries. Contrary to the statically scheduled Tile QR, we can not keep a part of the panel on the CPU thanks to an *a priori* task mapping. Instead we have to keep consistent the input and output data of CPU and GPU based kernels executing the same function, which sometimes introduces overhead by specific data initialization or data restoration.

Dynamic scheduling becomes, however, superior when the node is complex. First, on hybrid platforms with varying CPU / GPU ratios, it is difficult to perform an efficient schedule statically because of the heterogeneity of the processing units. Second, with the complexity of the memory hierarchies and buses, long-term predictions may not be accurate. Third, because the productivity is much higher when designing algorithms on top of a robust runtime system, it enables the implementation of more advanced

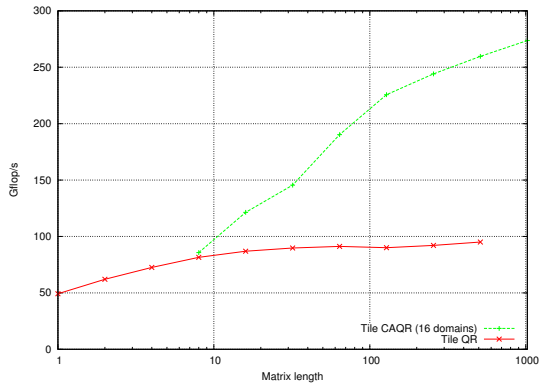

Figure 13. Performance of Tile CAQR on Nehalem-Quadro

algorithms. In the next Section, we describe how a complex algorithm such as the Tile CAQR can be easily implemented on top of StarPU.

VI. Tile CAQR: on top of StarPU

Our implementation of the Tile CAQR algorithm is based on a variation of the Tile QR algorithm previously designed in the context of multicore architectures [6]. We reused the four GPU and CPU kernels that we designed for the Tile QR algorithm (Section III) and a CPU version of the two additional kernels required for merging the domains.

Figure 13 shows the benefits of using the Tile CAQR algorithm on the Nehalem-Quadro platform when processing tall and skinny matrices. The considered matrices have indeed a small fixed number of columns (two blocks) and a large varying number of rows (x-axis). Since Tile QR performs the panel factorization in sequence, parallelism is very limited and the performance remains low even when matrices have a large number of rows (right-most part of Figure 13). On the other hand, with Tile CAQR algorithm, the panel is divided into multiple domains (16 here) that can be processed concurrently. When the matrix has a large number of rows, this latter approach enables us to extract parallelism from the panel factorization and achieves a much higher performance than standard Tile QR on our hybrid platform. In this study, we did not consider the use of GPU kernels for performing the reduction operations of the Tile CAQR algorithm. We expect that the incorporation of those kernels might further improve performance. This is future work.

VII. Conclusion

We have shown that we can efficiently exploit all resources of a multicore node enhanced with multiple GPUs to execute a central operation of dense linear algebra, the QR factorization. The first technical contribution is the design of efficient hybrid algorithms for implementing the GPU kernels. These kernels are furthermore highly tuned to achieve high performance. The second technical contribution is the implementation of advanced high-level algorithms (Tile QR and Tile CAQR) on top of two frameworks. We conclude from our study that static schedulers that perform optimizations thanks to an a priori knowledge of the mapping can achieve very high performance when the platform is relatively homogeneous (same number of GPUs and CPUs). However, when the node is more heterogeneous or when more complex algorithms need to be scheduled, this approach lacks of productivity. We have shown that a runtime system performing dynamic scheduling is then more appropriate to exploit the full potential of the machine and can even automatically show that it is close to theoretical upper bounds.

This work is a step towards exascale computing. However, many challenges remain. On architectures with important NUMA factors, it will be critical to design methods for efficiently supporting NUMA architectures. The extension to clusters and the related scalability issues are also to be investigated. We also plan to apply our method to other one-sided (such as LU decomposition) and two-sided (such as Hessenberg reduction) approaches. A first attempt to design efficient hybrid kernels for NVIDIA Fermi GPUs was not conclusive and thus not presented here. We plan to pursue this work to fully exploit the potential of this new generation of device. Expressing the algorithms in a high-level framework enables us to split the development of kernels from the design of scheduling policies. It has the advantages that kernels can be highly tuned and that advanced scheduling strategies may be employed. Furthermore, autotuning strategies will have to be designed to find the optimum blocking sizes and number of domains dependent upon the considered platform and upon the structure of the matrix.

Acknowledgment

The authors would like to thank Lionel Eyraud-Dubois for constructive discussions as well as Abdou Guermouche and Scott Wells for the review of a preliminary version.

References

[1] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM, 1997, iSBN: 0898713617.

[2] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, August 2002.

[3] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 2nd ed. Baltimore, MD, USA: The Johns Hopkins University Press, 1989.

[4] C. Bischof and C. van Loan, "The WY representation for products of Householder matrices," *J. Sci. Stat. Comput.*, vol. 8, pp. 2–13, 1987.

[5] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," Innovative Computing Laboratory, University of Tennessee, Tech. Rep., 2010.

[6] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra, "Tile QR Factorization with Parallel Panel Processing for Multicore Architectures," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium, Atlanta, GA, April 19-23*, 2010.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010.

[8] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra, "Comparative study of one-sided factorizations with multiple software packages on multi-core hardware," *2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '09)*, 2009.

[9] B. C. Gunter and R. A. van de Geijn, "Parallel out-of-core computation and updating the QR factorization," *ACM Transactions on Mathematical Software*, vol. 31, no. 1, pp. 60–78, 2005, DOI: 10.1145/1055531.1055534.

[10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1573–1590, 2008.

[11] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. V. Zee, and R. A. van de Geijn, "Scheduling of QR factorization algorithms on SMP and multi-core architectures," in *Proceedings of PDP'08*, 2008, FLAME Working Note #24.

[12] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," EECS Department, University of California, Berkeley, LAPACK Working Note 204, August 2008.

[13] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 7705, Apr. 2010.

[14] M. Anderson and J. Demmel, "Communication-avoiding QR decomposition for GPU," GPU Technology Conference, Research Poster A01, 2010.

[15] M. Fogué, F. D. Igual, E. S. Quintana-ortí, and R. V. D. Geijn, "Retargeting plapack to clusters with hardware accelerators flame working note #42," 2010.

[16] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, N. Saengpatsa, S. Tomov, and J. Dongarra, "A unified HPC environment for hybrid manycore/GPU distributed systems," LAPACK Working Note, Tech. Rep. 234, Oct. 2010.

[17] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862.

[18] G. F. Diamos and S. 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*. New York, NY, USA: ACM, 2008, pp. 197–200.

[19] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *ACM/IEEE SC'06 Conference*, 2006.

[20] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling Hierarchical N-body Simulations on GPU Clusters," in *Proceedings of the ACM/IEEE Supercomputing Conference 2010 (to appear)*, 2010.

[21] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense Linear Algebra Solvers for Multicore with GPU Accelerators," *Proc. of IPDPS'10*, 2010.

[22] S. Tomov, R. Nath, and J. Dongarra, "Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing," *Parallel Computing*, vol. DOI information: 10.1016/j.parco.2010.06.001, 2010.

[23] R. Nath, S. Tomov, and J. Dongarra, "An Improved MAGMA GEMM for Fermi GPUs," Innovative Computing Laboratory, University of Tennessee, Tech. Rep., 2010.

[24] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parellel Comput. Syst. Appl.*, vol. 27, no. 1-2, pp. 3–35, 2001, DOI: 10.1016/S0167-8191(00)00087-9.

[25] R. Vuduc, J. Demmel, and K. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proc. of SciDAC'05*, ser. Journal of Physics: Conference Series. San Francisco, CA: Institute of Physics Publishing, June 2005.

[26] J. Kurzak and J. J. Dongarra, "QR factorization for the CELL processor," *Scientific Programming, Special Issue: High Performance Computing with the Cell Broadband Engine*, vol. 17, no. 1-2, pp. 31–42, 2009.

[27] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurrency Computat.: Pract. Exper.*, vol. 21, no. 1, pp. 15–44, 2009, DOI: 10.1002/cpe.1467.

[28] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, Mar 2002.

[29] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*, Shangai, China, Dec. 2010, to appear.