Chapter 3

Dense Linear Algebra for Hybrid GPU-Based Systems

Stanimire Tomov
Department of Electrical Engineering and Computer Science, University of Tennessee

Jack Dongarra
Department of Electrical Engineering and Computer Science, University of Tennessee
Computer Science and Mathematics Division, Oak Ridge National Laboratory
School of Mathematics & School of Computer Science, Manchester University

3.1 Introduction

Since the introduction of multicore architectures, hardware designs are going through a renaissance due to the need for new approaches to manage the exponentially increasing:

1. Appetite for power, and
2. Gap between compute and communication speeds.

Hybrid graphics processing unit (GPU)-based multicore platforms, composed of both homogeneous multicores and GPUs, stand out among a confluence of current hardware trends as they provide an effective solution to these two challenges. Indeed, as power consumption is typically proportional to the cube of the frequency, GPUs have a clear advantage against current homogeneous

multicores, as GPUs' compute power is derived from many cores that are of low frequency. Furthermore, initial GPU experiences across academia, industry, and national research laboratories have provided a long list of success stories for specific applications and algorithms, often reporting speedups of order 10 to 100× compared to current x86-based homogeneous multicore systems [2].

3.1.1 Linear Algebra (LA)—Enabling New Architectures

Despite the current success stories involving hybrid GPU-based systems, the large-scale enabling of those architectures for computational science would still depend on the successful development of fundamental numerical libraries for them. Major issues in terms of developing new algorithms, programmability, reliability, and user productivity must be addressed. This chapter describes some of the current efforts toward the development of these fundamental libraries, and in particular, libraries in the area of dense linear algebra (DLA).

Historically, linear algebra has been in the vanguard of efforts to enable new architectures for computational science for good strategic reasons. First, a very wide range of science and engineering applications depend on linear algebra; these applications will not perform well unless linear algebra libraries perform well. Second, linear algebra has a rich and well understood structure for software developers to exploit algorithmically, so these libraries represent an advantageous starting point for the effort to bridge the yawning software gap that has opened up within the HPC community today.

3.1.2 MAGMA—LA Libraries for Hybrid Architectures

The Matrix Algebra on GPU and Multicore Architectures (MAGMA) project, and the MAGMA and MAGMA BLAS libraries [3] stemming from it, are used to demonstrate the techniques and their effect on performance. Designed to be similar to LAPACK in functionality, data storage, and interface, the MAGMA libraries will allow scientists to effortlessly port their LAPACK-relying software components and to take advantage of the new hybrid architectures. Current work targets GPU-based systems, and the efforts are supported by both government and industry, including NVIDIA, who recently recognized the University of Tennessee, Knoxville's (UTKs) Innovative Computing Laboratory (ICL) as a CUDA Center of Excellence. This is to further promote, expand, and support ICL's commitment toward developing **LA Libraries for Hybrid Architectures**.

Against this background, the main focus of this chapter will be to provide some high-level insight on **how to code/develop DLA for GPUs**. The approach described here is based on the idea that, in order to deal with the complex challenges stemming from the heterogeneity of the current GPU-based systems, optimal software solutions will themselves have to hybridize, combining the strengths of the system's hybrid components. That is, *hybrid algorithms* that match algorithmic requirements to the architectural strengths

of the system's hybrid components must be developed. It has been shown that properly designed hybrid algorithms for GPU-based multicore platforms work for the core DLA routines, namely the one- and two-sided matrix factorizations.

3.2 Hybrid DLA Algorithms

The development of high-performance DLA for homogeneous multicores has been successful in some cases, like the one-sided factorizations, and difficult for others, like the two-sided factorizations. The situation is similar for GPUs—some algorithms map well, others do not. Developing algorithms for a combination of these two architectures (to use both multicore and GPUs) though can be beneficial and should be exploited, especially since in many situations, the computational bottlenecks for one of the components (of this hybrid system) may not be for the other. Thus, developing **hybrid algorithms** that properly split and schedule the computation over different hardware components may lead to very efficient algorithms. The goal is to develop these new, hybrid algorithms for the area of DLA, and moreover, show that the new developments:

- Leverage prior DLA developments, and
- Achieve what has not been possible so far, e.g., using just homogeneous multicores.

3.2.1 How to Code DLA for GPUs?

The question of how to code for any architecture, including GPUs, is complex, e.g., involving issues in terms of choosing a language, programming model, developing new (GPU specific) algorithms, programmability, reliability, and user productivity. Nevertheless, it is possible to give some major considerations and directions that have already shown promissing results:

Use CUDA / OpenCL CUDA is currently the language of choice for programming GPUs. It facilitates a data-based parallel programming model that has turned out to be a remarkable fit for many applications. Moreover, current results show its programming model allows applications to scale on many cores. DLA is no exception as performance relies on the performance of Level 2/3 BLAS—essentially a data parallel set of subroutines that are scaling on current many-core GPUs (see also Chapter 4). The approach described here also shows how the BLAS scalability is in fact translated into scalability on higher level routines (LAPACK).

Similarly to CUDA, OpenCL also has its roots in the data-based parallelism (now both moving to support task-based parallelism). OpenCL is still yet to be established but the fact that it is based on a programming model with already established potential and the idea of providing portability—across heterogeneous platforms consisting of CPUs, GPUs, and other processors—makes it an excellent candidate for coding hybrid algorithms.

Use GPU BLAS Performance of DLA critically depends on the availability of fast BLAS, especially on the Level 3 BLAS matrix-matrix multiplication. Older generation GPUs did not have memory hierarchy and their performance exclusively relied on high bandwidth. Therefore, although there has been some work in the field, the use of older GPUs has not led to significantly accelerated DLA. For example, Fatahalian et al. studied SGEMM and their conclusion was that CPU implementations outperform most GPU implementations. Similar results were produced by Galoppo et al. on LU factorization. The introduction of **memory hierarchy** in current GPUs though has drastically changed the situation. Indeed, by having memory hierarchy, GPUs can be programmed for memory reuse and hence not rely exclusively on their high bandwidth. An illustration of this fact is given in Figure 3.1, showing the performance of correspondingly a compute-bound (matrix-matrix multiplication on the left) and a memory-bound routine (matrix-vector multiplication on the right). Having fast BLAS is significant because algorithms for GPUs can now leverage prior DLA developments, which have traditionally relied on fast BLAS. Of course there are GPU-specific optimizations, as will be shown, like performing extra-operations, BLAS fusion, etc, but the important fact is, high-performance algorithms can be coded at a high level, just using BLAS, often abstracting the developer from the need of low-level GPU-specific coding.

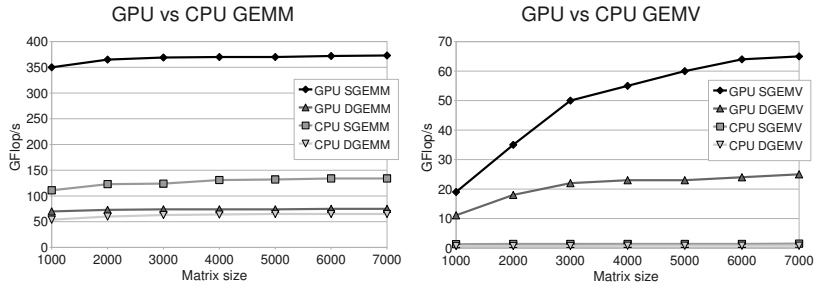


FIGURE 3.1: BLAS on GPU (GTX 280) *vs* CPU (8× Intel Xeon 2.33GHz).

Use Hybrid Algorithms Current GPUs feature massive parallelism but serial kernel execution. For example NVIDIA's GTX280 has 30 multipro-

cessors, each multiprocessor having eight SIMD functional units, each unit capable of executing up to three (single floating point) operations per cycle. At the same time, kernels are executed serially; only one kernel is allowed to run at a time using the entire GPU. This means that only large, highly parallelizable kernels can run efficiently on GPUs. The idea of using hybrid algorithms presents an opportunity to remedy this situation and therefore enable the efficient use of GPUs well beyond the case of data-parallel applications. Namely, the solution and advice to developers is to use a hybrid coding approach, where small, non-parallelizable kernels would be executed on the CPU, and only large, parallelizable kernels on the GPU. Although GPUs move towards supporting task-based parallelism as well (e.g., advertised for the next generation NVIDIA GPUs, code named "Fermi" [4]), small tasks that arise in DLA would still make sense to be done on the CPU for various reasons, e.g., to use the x-86 software infrastructure. Moreover, efficient execution would still require parallelism and small tasks still may be difficult or impossible to parallelize.

3.2.2 The Approach—*Hybridization of DLA Algorithms*

The above considerations are incorporated in the following *Hybridization of DLA Algorithms* approach:

- Represent DLA algorithms as a collection of BLAS-based tasks and dependencies among them (see Figure 3.2):
  - Use parametrized task granularity to facilitate auto-tuning;
  - Use performance models to facilitate the task splitting/mapping.
- Schedule the execution of the BLAS-based tasks over the multicore and the GPU:
  - Schedule small, non-parallelizable tasks on the CPU and large, parallelizable on the GPU;
  - Define the algorithm's *critical path* and prioritize its execution/scheduling.

The splitting of the algorithms into tasks is in general easy, as it is based on the splitting of large BLAS into smaller ones. More challenging is choosing the granularity and shape of the splitting and the subsequent scheduling of the sub-tasks. There are two main guiding directions on how to design the splitting and scheduling of tasks. First, the splitting and scheduling should allow for asynchronous execution and load balance among the hybrid components. Second, it should harness the strengths of the components of a hybrid architecture by properly matching them to algorithmic/task requirements. Examples demonstrating these general directions are given in the next two sections.

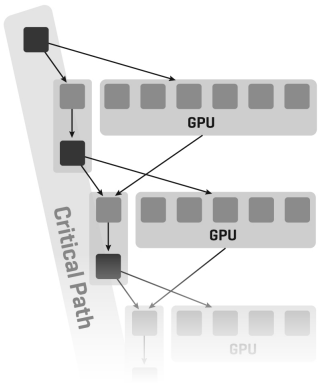


FIGURE 3.2: Algorithms as a collection of BLAS-based tasks and dependencies among them (DAGs) for hybrid GPU-based computing.

Next, choosing the task granularity can be done by parametrizing the tasks' sizes in the implementations and tuning them empirically [7]. The process can be automated [13], often refered to as *auto-tuning*. Auto-tuning is crucial for the performance and the maintenance of modern numerical libraries, especially for hybrid architectures and algorithms for them. Figuratively speeking, it can be regarded as both *the Beauty and the Beast* behind hybrid DLA libraries (e.g., MAGMA) as it is an elegant and very practical solution for easy maintenance and performance portability, while often being a brute force, empirically based exhaustive search that would find and set automatically the best performing algorithms/kernels for a specific hardware configuration. The "exhaustive" search is often relaxed by applying various performance models.

Finally, the problem of scheduling is of crucial importance for the efficient execution of an algorithm. In general, the execution of the critical path of an algorithm should be scheduled as soon as possible. This often remedies the problem of synchronizations introduced by small, non-parallelizable tasks (often on the critical path; scheduled on the CPU) by overlapping their execution with the execution of larger more parallelizable ones (often Level 3 BLAS; scheduled on the GPU).

These principles are general enough to be applied in areas well beyond DLA. Usually they come with specifics, induced by the architecture and the algorithms considered. The following two sections present some of these specifics for, correspondingly, the classes of one-sided and two-sided dense matrix factorizations.

3.2.3 One-Sided Factorizations

This section describes the hybridization of LAPACK's one-sided factorizations—LU, QR, and Cholesky—on dense matrices. These factorizations are important because they are the first of two steps in solving dense linear systems of equations. The factorization represents the bulk of the computation—$O(N^3)$ floating point operations in the first step *vs* $O(N^2)$ in the second step—and therefore has to be highly optimized. LAPACK uses block-partitioned algorithms, and the corresponding hybrid algorithms are based on them.

The opportunity for acceleration using hybrid approaches (CPU and GPU) has been noticed before in the context of one-sided factorizations. In particular, while developing algorithms for GPUs, several groups observed that panel factorizations are often faster on the CPU than on the GPU, which led to the development of highly efficient, one-sided hybrid factorizations for a single CPU core and a GPU [6,7,9], multiple GPUs [6,10], and multicore+GPU systems [11]. M. Fatica [12] developed hybrid DGEMM and DTRSM for GPU-enhanced clusters and used them to accelerate the Linpack benchmark. This approach, mostly based on BLAS level parallelism, results only in minor or no modifications to the original source code.

MAGMA provides two interfaces to the hybrid factorizations. These are the CPU interface, where the input matrix is given in the CPU memory and the result is expected also in the CPU memory, and the GPU interface, where the input and the output are on the GPU memory. In both cases the bulk of the computation is done on the GPU and along the computation; the submatrices that remain to be factored always reside on the GPU memory. Panels are copied to the CPU memory, processed on the CPU using LAPACK, and copied back to the GPU memory (see Figure 3.3). Matrix updates are done on the GPU. Thus, for any panel transfer of size $n \times nb$ elements, a sub-matrix of size $n \times (n - nb)$ is updated on the GPU (e.g., this is the case for the right-looking versions of LU and QR; for the left-looking Cholesky the ratios are $nb \times nb$ elements transfered *vs* $n \times nb$ elements updated), where nb is a blocking size and n is the size of the sub-matrix that remains to be factored. These ratios of communications *vs* computations allow for mostly overlapping the panel factorizations on the CPU with updates on the GPU (see below the specifics). Figure 3.3 illustrates this typical pattern (as just described) of hybridization for the one-sided factorizations in the GPU interface.

The CPU interface can reuse the GPU interface implementation by wrapping it around two matrix copies—one at the beginning copying the input matrix from the CPU to the GPU memory, and one at the end copying the final result from the GPU back to the CPU memory. This overhead can be partially avoided, though. The input matrix can be copied to the GPU by starting an asynchronous copy of the entire matrix except the first panel and overlapping this copy with the factorization of the first panel on the CPU. The GPU-to-CPU copy at the end of the factorizations can be replaced by a

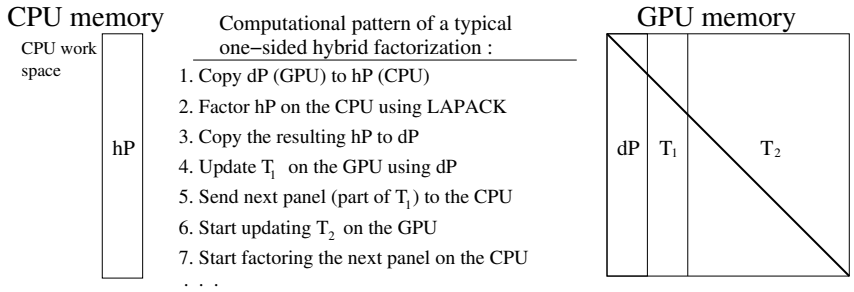


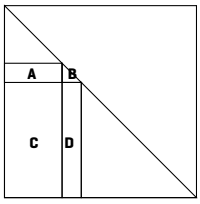
FIGURE 3.3: A typical hybrid pattern of computation and communication for the one-sided matrix factorizations in MAGMA's GPU interface.

continuous accumulation of the output directly on the CPU. One way to do it is by sending the entire block of columns containing the current panel (*vs* just the panel) and keeping copies of the factored panels on the CPU. This way, as the computation progresses, the final result is accumulated on the CPU as a byproduct of the computation, avoiding the overhead of an entire matrix copy.

Below are given some of the specifics in the development of the hybrid Cholesky, QR, and LU factorizations, respectively:

Cholesky Factorization MAGMA uses the left-looking version of the block Cholesky factorization. Figure 3.4 shows an iteration of the standard block Cholesky algorithm coded in correspondingly MATLAB and LAPACK style, and how these standard implementations can easily be translated into a hybrid implementation. Note the simplicity and the similarity of the hybrid code with the LAPACK code. The only difference is the two CUDA calls needed to offload data back and forth from the CPU to the GPU. Also, note that steps (2) and (3) are independent and can be overlapped—step (2), a Cholesky factorization task on a small diagonal block, is scheduled on the CPU using a call to LAPACK and step (3), a large matrix-matrix multiplication, on the GPU. This is yet another illustration of the general guidelines mentioned in the previous two sections. The performance of this algorithm is given in Section 3.3.

QR Factorization Currently, MAGMA uses static scheduling and a right-looking version of the block QR factorization. The panel factorizations are scheduled on the CPU using calls to LAPACK, and the Level 3 BLAS updates on the trailing sub-matrices are scheduled on the GPU. The trailing matrix updates are split into 2 parts—one that updates just the next panel and a second one updating the rest, i.e., correspondingly sub-matrices T_1 and T_2 as given in Figure 3.3. The next panel update (i.e., T_1) is done first, sent to the CPU, and the panel factorization on the CPU is overlapped with the second part of the trailing matrix



MATLAB code	LAPACK code	Hybrid code
(1) B = B – A*A'	ssyrk_("L", "N", &nb, &j, &mone, hA[j,0], ...)	cublasSsyrk('L', 'N', nb, j. mone, dA[j,0], ...)
		cublasGetMatrix(nb, nb, 4, dA[j, j], *lda, hwork, nb)
(2) B = chol(B, 'lower')	spotrf_("L", &nb, hA[j, j], lda, info)	cublasSgemm('N', 'T', j, ...)
(3) D = D – C*A'	sgemm_("N", "T", &j, ...)	spotrf_("L", &nb, hwork, &nb, info)
		cublasSetMatrix(nb, nb, 4, hwork, nb, dA[j, j], *lda)
(4) D = D\B	strsm_("R", "L", "T", "N", &j, ...)	cublasStrsm('R', 'L', 'T', 'N', j, ...)

FIGURE 3.4: Pseudo-code implementation of the hybrid Cholesky. `hA` and `dA` are pointers to the matrix to be factored correspondingly on the host (CPU) and the device (GPU).

(i.e., T_2). This technique is known as *look-ahead*, e.g., used before in the Linpack benchmark [13]. Its use enables the overlap of CPU and GPU work (and some communications). Figure 3.5 illustrates this by quantifying the CPU-GPU overlap for the case of QR in single precision arithmetic. Note that, in this case for matrices of size above $6,000$, the

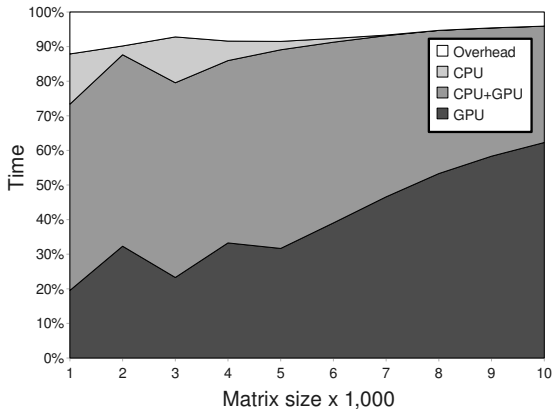


FIGURE 3.5: Time breakdown for the hybrid QR from MAGMA in single precision arithmetic on GTX280 GPU and Intel Xeon 2.33GHz CPU.

CPU work on the panels is entirely overlapped by work on the GPU.

It is important to note that block factorizations (both one and two sided) inherently lead to multiplications with triangular matrices. To execute them in data-parallel fashion on GPUs, it is often more efficient to put zeroes in the unused triangular parts and perform the multiplication as having general dense matrices. To give an example, the block QR algorithm accumulates orthogonal Level 2 BLAS transformations during the panel factorization. The accumulated transformation has the form

$$Q_i = I - V_i T_i V_i^T$$

where V_i is of size $k \times nb$, $k \geq nb$ and T_i is $nb \times nb$. This transformation is than applied as a Level 3 BLAS to the trailing sub-matrix. The LAPACK implementation splits the multiplication with V_i into two Level 3 BLAS calls, as the top $nb \times nb$ sub-matrix of V_i is triangular. A GPU implementation is more efficient if the multiplication with V_i is performed as one BLAS call, and therefore is enabled, e.g., by providing an efficient mechanism to put zeroes in the unused triangular part of V_i (before the multiplication) and restoring the original (after the multiplication).

This is just one example of a GPU-specific optimization technique where to get higher performance extra flops must be done or separate kernels must be fused into one.

LU Factorization Similarly to QR, MAGMA uses a right-looking version of the LU factorization. The scheduling is static using the look-ahead technique. Interchanging rows of a matrix stored in column major format, needed in the pivoting process, cannot be done efficiently on current GPUs. We use the LU factorization algorithm by V. Volkov and J. Demmel [6] that removes the above mentioned bottleneck. The idea behind it is to transpose the matrix in the GPU memory. This is done once at the beginning of the factorization so that row elements are contiguous in memory, i.e., equivalent to changing the storage format to row major. Row interchanges now can be done efficiently using coalescent memory accesses on the GPU (*vs* strided memory accesses for a matrix in column major format). The panels are being transposed before being sent to the CPU for factorization, i.e., moved back to the standard for LAPACK column major format. Compared to the non-transposed version, this algorithm runs approximately 50% faster on current NVIDIA GPUs, e.g., GTX 280.

3.2.4 Two-Sided Factorizations

If the importance of the one-sided factorizations stems from their role in solving linear systems of equations, the importance of the two-sided factorizations stems from their role in solving eigenvalue problems. The two-sided factorizations are an important first step in solving eigenvalue problems. Similarly

to the one-sided, the two-sided factorizations are compute intensive ($O(N^3)$ flops) and therefore also have to be highly optimized.

The two-sided factorizations can be organized as block-partitioned algorithms. This is used in LAPACK to develop efficient signle CPU/core algorithms and can be used as the basis for developing hybrid algorithms. The development follows the main approach already described. This section will describe the specifics involved and will actually demonstrate much higher speedups of accelerating two-sided factorizations *vs* the speedups in accelerating the one-sided factorizations.

The hybridization of the two-sided factorizations can be best explained with the reduction to upper Hessenberg form, denoted further by HR, which is stressed in this section. The operation count for the reduction of an $N \times N$ matrix is approximately $\frac{10}{3}N^3$, which, in addition to not running efficiently on current architectures, makes the reduction a very desirable target for acceleration.

The bottleneck: The problem of accelerating the two-sided factorization algorithms stems from the fact that these algorithms are rich in Level 2 BLAS operations, which do not scale on multicore architectures and actually run only at a fraction of the machine's peak performance. This is shown in Figure 3.1, right. There are dense linear algebra (DLA) techniques that can replace Level 2 BLAS operations with Level 3 BLAS. These are the block algorithms, already mentioned several times, where the application of consecutive Level 2 BLAS operations that occur in the algorithms can be delayed and accumulated so that at a later moment the accumulated transformation is applied at once as a Level 3 BLAS (see LAPACK [14] and the specifics related to QR above). This approach totally removes Level 2 BLAS flops from Cholesky and reduces its amount to $O(n^2)$ flops in LU and QR thus making it asymptotically insignificant compared to the total $O(n^3)$ amount of operations for these factorizations. The same technique can be applied to HR [15], but in contrast to the one-sided factorizations, it still leaves about 20% of the total number of operations as Level 2 BLAS. Note that 20% of Level 2 BLAS is significant because, in practice, using a single core of a multicore machine, this 20% can take about 70% of the total execution time, thus leaving the grim perspective that multicore use—no matter how many cores would be available—can ideally reduce only the 30% of the execution time that is spent on Level 3 BLAS.

Bottleneck identification: For large applications, tools like TAU [16] can help in locating performance bottlenecks. TAU can generate execution profiles and traces that can be analyzed with other tools like ParaProf [17], Jumpshot [18], and Kojak [19]. Profiles of the execution time (or of PAPI [20] hardware counters) on runs using various numbers of cores of a multicore processor can be compared using ParaProf to easily see which functions scale, what is the performance for various parts of the code, what percentage of the execution time is spent in the various functions, etc. In the case of HR, using a dual socket quad-core Intel Xeon at 2.33GHz, this analysis easily identifies a call

to a single matrix-vector product that runs at about 70% of the total time, does not scale with multicore use, and has only 20% of the total amount of flops. Algorithmically speaking, this matrix-vector product is part of the panel factorization, where the entire trailing matrix (denoted by A_j in Figure 3.6) has to multiply a currently computed Householder reflector vector v_j (see [10] for further datail on the algorithm). These findings are also illustrated in Figure 3.6.

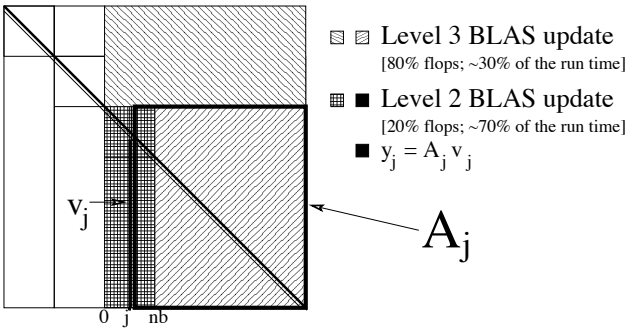


FIGURE 3.6: HR computational bottleneck: Level 2 BLAS $y_j = A_j v_j$.

Since fast Level 2 BLAS are available for GPUs (Figure 3.1, right), it is clear that the operation has to be scheduled for execution on the GPU. In other words, having fast implementations for all kernels, the development of the hybrid HR algorithms is now just a matter of splitting the computation into tasks and properly scheduling the tasks' execution on the available hybrid hardware components. Below are the details of the main steps in this process of *hybridization* of the HR algorithm:

Task Spliting Studying the execution and data flow of an algorithm is important in order to properly split the computation into tasks and dependencies. It is important to study the memory footprint of the routines, e.g., what data are accessed and what data are modified. Moreover, it is important to identify the algorithm's critical path and to decouple from it as much work as possible, as the tasks outside the critical path in general would be the ones trivial to parallelize. Applied to HR, this analysis identifies that the computation must be split into three main tasks, further denoted by P_i, M_i, and G_i, where i is iteration index for the block HR and is described as follows. The splitting is motivated by and associated with operations updating three corresponding matrices, for convenience denoted by the name of the task updating them. The splitting is illustrated in Figure 3.7, left, and described as follows:

- The panel factorization task P_i (20% of the flops) updates the current panel and accumulates matrices V_i, T_i, and Y_i needed for the trailing matrix update;

- Task G_i (60% of the flops) updates the trailing sub-matrix

$$G_i = (I - V_i\ T_i\ V_i^T)\ G_i\ (I - V_i\ T_i\ V_i(nb+1:\ ,\ :\ )^T)$$

- Task M_i (20% of the flops) updates the sub-matrix, determined to fall **outside of the critical path** of the algorithm

$$M_i = M_i\ (I - V_i\ T_i\ V_i^T).$$

.

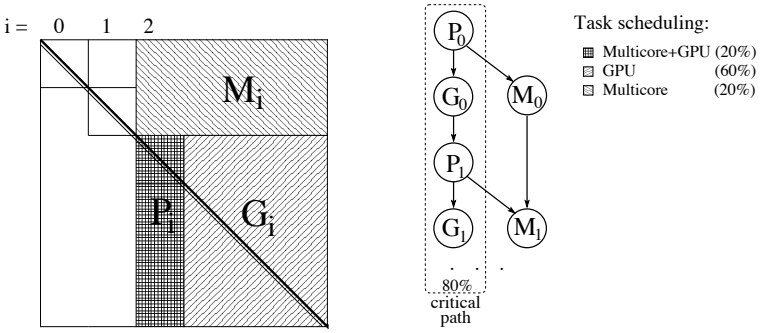


FIGURE 3.7: Main tasks and their scheduling.

We note that this splitting is motivated by the memory footprint analysis. Using this particular splitting, one can see that task M_i gets independent of G_i and falls outside of the critical path of the algorithm (illustrated in Figure 3.7, right). This is important for scheduling the tasks' execution over the components of the hybrid system. Note that the critical path is still 80% of the total amount of flops.

Task Scheduling The scheduling is given also in Figure 3.7, right. The tasks on the critical path must be scheduled as fast as possible—and the scheduling must be hybrid, using both the Multicore and the GPU. The memory footprint of task P_i, with 'P' standing for panel, is both P_i and G_i, but G_i is accessed only for the time consuming computation of $y_j = A_j v_j$ (see Figure 3.6). Therefore, the part of P_i that is constrained to the panel (not rich in parallelism, with flow control statements) is scheduled on the multicore using LAPACK, and the time consuming $y_j = A_j v_j$ (highly parallel but requiring high bandwidth) is scheduled on the GPU. G_i, with 'G' standing for GPU, is scheduled on the GPU. This is a Level 3 BLAS update and can be done very efficiently on the GPU. Moreover, note that G_{i-1} contains the matrix A_j needed for task P_i, so for the computation of $A_j v_j$ we have to only send v_j to the GPU and the resulting y_j back from the GPU to the multicore. The scheduling so far heavily uses the GPU, so in order to simultaneously make the

critical path execution faster and to make a better use of the multicore, task M_i, with 'M' standing for multicore, is scheduled on the multicore.

Hybrid HR Figure 3.8 illustrates the communications between the multicore and GPU for i^{th} iteration (outer) of the hybrid block HR algorithm. Shown is the j^{th} inner iteration. Note that, as in the case of the one-sided factorizations, the matrix to be factored resides originally on the GPU memory, and as the computation progresses the result is continuously accumulated on the CPU. At the end, the factored matrix is available on the CPU memory in a format identical to LAPACK's. The tasks scheduling is as given above.

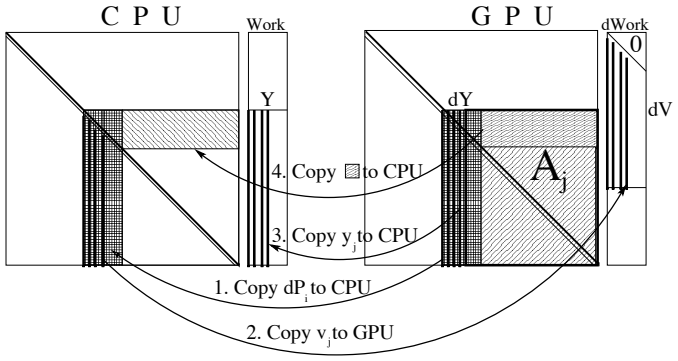


FIGURE 3.8: CPU/GPU communications for inner/outer iteration j/i.

This approach can be applied to the rest of the two-sided matrix factorizations. The key component—having fast GPU implementations of the various Level 2 BLAS matrix-vector products that may be needed in the different algorithms—is now available through the MAGMA BLAS library (see Chapter 4). Having the kernels needed, as stated above, it is now simply a matter of organizing the computation in a hybrid fashion over the available hardware components.

3.3 Performance Results

The performance results provided in this section use NVIDIA's GeForce GTX 280 GPU and its multicore host, a dual socket quad-core Intel Xeon running at 2.33GHz. Kernels executed on the multicore use LAPACK and BLAS from MKL 10.1, and BLAS kernels executed on the GPU use a combination of CUBLAS 2.1 and MAGMA BLAS 0.2, unless otherwise noted.

The performance of the hybrid Cholesky factorization is given in Figure

3.9. It runs asymptotically at 300 Gflop/s in single and at almost 70 Gflop/s in double precision arithmetic.

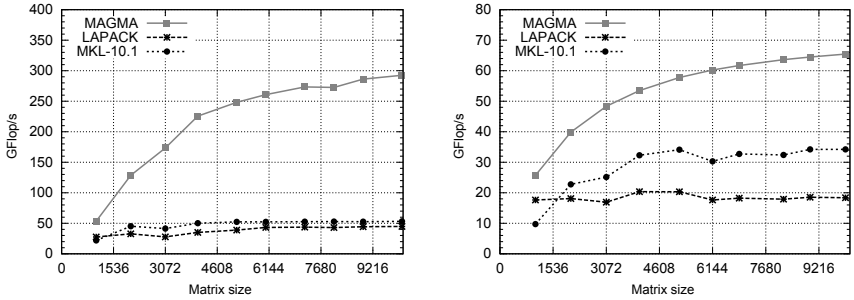


FIGURE 3.9: Performance of MAGMA's hybrid Cholesky in single (left) and double precision (right) arithmetic on GTX 280 *vs* MKL 10.1 and LAPACK (with multi-threaded BLAS) on Intel Xeon dual socket quad-core 2.33GHz.

The performance of the hybrid QR factorization is given in Figure 3.10. It runs asymptotically almost at 290 Gflop/s in single and at almost 68 Gflop/s in double precision arithmetic.

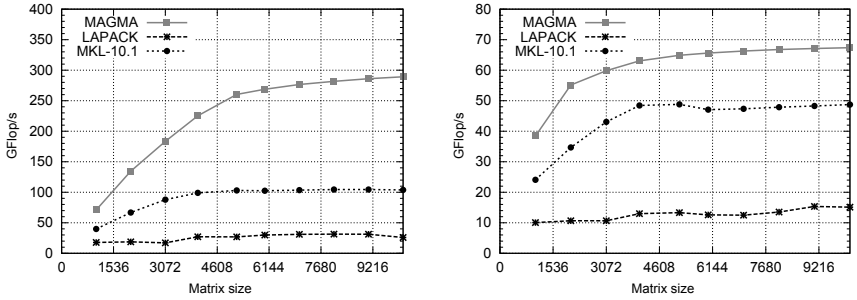


FIGURE 3.10: Performance of MAGMA's hybrid QR in single (left) and double precision (right) arithmetic on GTX 280 *vs* MKL 10.1 and LAPACK (with multi-threaded BLAS) on Intel Xeon dual socket quad-core 2.33GHz.

The performance of the hybrid LU factorization is given in Figure 3.11. It runs asymptotically almost at 320 Gflop/s in single and at almost 70 Gflop/s in double precision arithmetic.

Figure 3.12 shows the performance of two versions of the hybrid HR algorithms in double precision arithmetic. The performance is also compared to the block HR on single core and multicore. The basic hybrid HR is for one core accelerated with one GPU. The "Multicore+GPU" hybrid algorithm is the one described in this chapter where tasks M_i are executed on the available CPU cores. The result shows an enormous speedup of 16× when compared to the current block HR running on just homogeneous multicore. The techniques

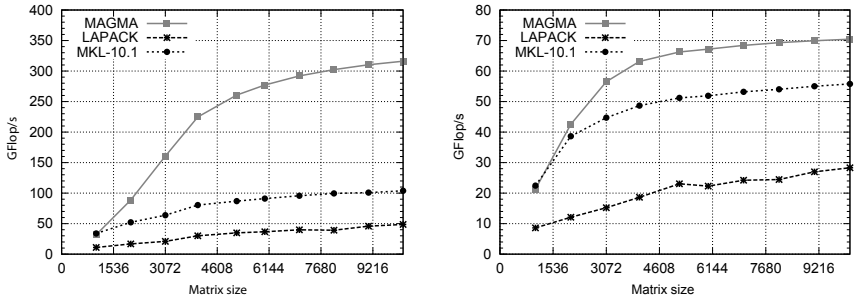


FIGURE 3.11: Performance of MAGMA's hybrid LU in single (left) and double precision (right) arithmetic on GTX 280 *vs* MKL 10.1 and LAPACK (with multi-threaded BLAS) on Intel Xeon dual socket quad-core 2.33GHz.

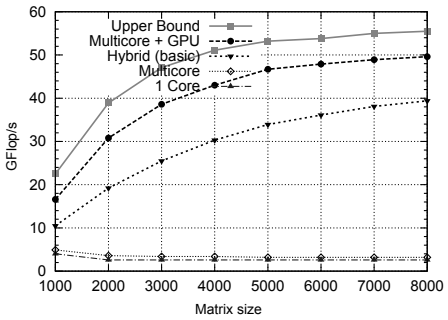


FIGURE 3.12: Performance of the hybrid HR in double precision (bottom) arithmetic on GTX 280 *vs* MKL 10.1 on Intel Xeon dual socket quad-core 2.33GHz.

in the basic implementation account for most of the acceleration, which is due to the use of hybrid architectures and the proper algorithmic design—splitting the computation into tasks and their scheduling so that we match algorithmic requirements to architectural strengths of the hybrid components.

This performance gets to be asymptotically within 90% of the "upper bound" performance, as shown in Figure 3.12. Here upper bound denotes the performance of just the *critical path* of the algorithm when no synchronizations and data transfer times are taken into account, i.e., this is the performance of tasks P_i and G_i (without counting M_i) just based on the performance of the BLAS used.

3.4 Summary

This work is on the development of numerical linear algebra libraries and is motivated by hardware changes that have rendered legacy numerical libraries inadequate for the new architectures. GPUs, homogeneous multicore architectures, and hybrid systems based on them adequately address the main requirements of new architectures—to keep power consumption and the gap between compute and communication speeds low. There is every reason to believe that future systems will continue the general trend taken by GPUs and hybrid combinations of GPUs with homogeneous multicores—freeze the frequency and keep escalating the number of cores; stress on data-parallelism to provide high bandwidth to the escalating number of cores—and thus render this work relevant for future system designs.

This chapter presented a concept on how to code/develop dense linear algebra algorithms for GPUs. The approach is general enough to be applicable to areas well beyond dense linear algebra and is implemented on a high-enough level to guarantee easy portability and relevance for future hybrid systems. In particular, the approach uses CUDA to develop low-level kernels when needed, but mostly relies on high-level libraries like LAPACK for CPUs and BLAS for CPUs and GPUs. Moreover, the approach is based on the development of hybrid algorithms, where in general small, non-parallelizable tasks are executed on the CPU, reusing the existing software infrastructure for standard architectures, and large, data-parallel tasks are executed on the GPU.

It was shown that using this approach in the area of dense linear algebra, one can leverage prior DLA developments and achieve what has not been possible so far, e.g., using just homogeneous multicore architectures. In particular, the approach uses LAPACK to execute small, non-parallelizable tasks on the CPU. Tasks that are bottlenecks for the CPU, like Level 2 BLAS, are properly designed in the new algorithms and scheduled for execution on the GPU.

Specific examples were given on fundamental dense linear algebra algorithms. Namely, it was demonstrated how to develop hybrid one-sided and two-sided matrix factorizations. Although the one-sided factorizations can be represented as Level 3 BLAS and ran efficiently on current multicore architectures, GPUs still can accelerate them significantly, depending on the hardware configuration, e.g., $O(1)\times$ for the systems used in the numerical experiments presented. In the case of two-sided factorizations, the speedups are even higher, e.g., $O(10)\times$, as implementations using homogeneous multicores, no matter how many cores are available, currently run them at the speed of a single core.

The implementations are now freely available through the MAGMA library site http://icl.eecs.utk.edu/magma/.

Bibliography

[1] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. http://developer.download.nvidia.com, 2007.

[2] General-purpose computation using graphics hardware. http://www.gpgpu.org.

[3] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 Users' Guide. http://icl.cs.utk.edu/magma, November 2009.

[4] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/object/fermi_architecture.html, 2009.

[5] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing* 27(1-2):2001, 3–35.

[7] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *Proceedings of HPCMP Users Group Conference 2008*, July 14-17 2008, http://www.cs.utk.edu/~tomov/ugc2008_final.pdf.

[8] V. Volkov and J. 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, 2008. IEEE Press.

[9] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. LAPACK Working Note 200, May 2008.

[10] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. A scalable high performant cholesky factorization for Multicore with GPU Accelerators. LAPACK Working Note 223, November 2009.

[11] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. LAPACK Working Note 210, October 2008.

[12] M. Fatica Accelerating Linpack with CUDA on heterogenous clusters. GPGPU-2, pages 46–51, Washington, DC, 2009.

[13] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:820, 2003, pp. 1–18.

[14] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. `http://www.netlib.org/lapack/lug/`.

[15] S. Hammarling, D. Sorensen, and J. Dongarra. Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comput. Appl. Math* 27:1987, 215–227.

[16] S. Shende and A. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.* 20(2):2006, 287–311.

[17] R. Bell, A. Malony, and S. Shende. ParaProf: A portable, extensible, and scalable tool for parallel performance profile analysis Euro-Par, 2003, pp. 17–26.

[18] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *HPC Applications* 13(2):1999, 277–288.

[19] F. Wolf and B. Mohr. Kojak—A tool set for automatic performance analysis of parallel applications. Proceedings of Euro-Par 2003 Klagenfurt, Austria, August 26–29, 2003, pp. 1301–1304.

[20] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. Proc. of DoD HPCMP u&c pp. 7–10.

[21] S. Tomov and J. Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. Technical Report 219, LAPACK Working Note 219, May 2009.