LU Factorization with Partial Pivoting for a Multicore System with Accelerators

Jakub Kurzak, *Member*, *IEEE*, Piotr Luszczek, *Member*, *IEEE*, Mathieu Faverge, *Member*, *IEEE*, and Jack Dongarra, *Life Fellow*, *IEEE*

Abstract—LU factorization with partial pivoting is a canonical numerical procedure and the main component of the high performance LINPACK benchmark. This paper presents an implementation of the algorithm for a hybrid, shared memory, system with standard CPU cores and GPU accelerators. The difficulty of implementing the algorithm for such a system lies in the disproportion between the computational power of the CPUs, compared to the GPUs, and in the meager bandwidth of the communication link between their memory systems. An additional challenge comes from the complexity of the memory-bound and synchronization-rich nature of the panel factorization component of the block LU algorithm, imposed by the use of partial pivoting. The challenges are tackled with the use of a data layout geared toward complex memory hierarchies, autotuning of GPU kernels, fine-grain parallelization of memory-bound CPU operations and dynamic scheduling of tasks to different devices. Performance in excess of one TeraFLOPS is achieved using four AMD Magny Cours CPUs and four NVIDIA Fermi GPUs.

Index Terms—Gaussian elimination, LU factorization, partial pivoting, multicore, manycore, GPU, accelerator

1 Introduction

This paper presents an implementation of the canonical formulation of the LU factorization, which relies on partial pivoting for numerical stability. It is equivalent to the DGETRF function from the LAPACK numerical library [1]. Since the algorithm is coded in double precision, it can serve as the basis for an implementation of the high performance LINPACK benchmark (HPL) [13]. The target platform is a system combining one CPU board with four 12-core CPUs and one GPU board with four 14-core GPUs, for the total number of 104 hybrid cores. Here, *GPU core* means a device that can independently schedule instructions, which in NVIDIA nomenclature is called a *streaming multiprocessor*. It is not to be confused with a *CUDA core*. The memory system of the CPUs, referred to as the *host memory* is a *cache-coherent nonuniform memory access* shared memory system. The GPUs have their private memories, referred to as *device memories*. Communication between the host memory and the device memories is handled by *direct memory access* (DMA) engines of the GPUs and crosses the *PCI express* bus.

Numerous challenges are posed both by the target hardware and the target algorithm. Although presenting a similar number of cores, the GPUs have an order of magnitude higher floating-point peak performance. The disproportion is exacerbated by the fact that GPUs are tasked with regular, data-parallel and compute intensive work, while CPU are tasked with irregular, synchronization-rich and memory-bound work. The algorithm itself is challenging, specifically the technique of partial pivoting, which introduces irregular processing patterns and hard synchronization points. These challenges are tackled with a combination of both well established and novel techniques in parallel dense linear algebra, such as:

- tile matrix layout,
- GPU kernel autotuning,
- parallel recursive panel factorization,
- the technique of lookahead,
- dynamic (superscalar) task scheduling,
- communication and computation overlapping.

Notably, the level of performance reported in this work could be accomplished thanks to recently developed capabilities, such as a GPU kernel autotuning methodology and superscalar scheduling techniques.

1.1 Motivation

Two trends can be clearly observed in microprocessor technology: steadily increasing number of cores and integration of hybrid cores in a single chip. Current commodity processors go as high as 16 cores (e.g., AMD Interlagos) and all major microprocessor companies develop hybrid chips (NVIDIA Tegra, AMD Fusion, Intel MIC). It is to be expected, then, that in a few years hybrid chips with O(100) cores will be the norm, which is why the platform of choice for this paper is a system with 104 cores, 48 classic superscalar cores and 56 accelerator (GPU) cores. At the same time, accelerators are steadily gaining traction in many areas of scientific computing [2], [19], [21], [32].

Although the HPL is commonly perceived as an artificial benchmark, there are actually numerous examples of applications relying heavily on the Gaussian elimination for the solution of a large dense system of linear equations. For instance, the electromagnetics community is a major user of dense linear solvers. Of particular interest is the

• *The authors are with the Department of Electrical Engineering and Computer Science, University of Tennessee, 1122 Volunteer Blvd, Ste 203 Claxton, Knoxville, TN 37996.*
E-mail: {kurzak, luszczek, faverge, dongarra}@eecs.utk.edu.

Manuscript received 18 Jan. 2012; revised 1 Aug. 2012; accepted 5 Aug. 2012; published online 22 Aug. 2012.
Recommended for acceptance by B. de Supinski.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-01-0038.
Digital Object Identifier no. 10.1109/TPDS.2012.242.

computation of the *radar cross section* of an aircraft or a sea vessel. The problem is commonly solved using the *boundary element method*, sometimes also referred to as the *method of moments*, which produces a large dense system of linear equations, and employs the Gaussian elimination for its solution.

The problem of achieving a self-sustaining fusion reaction provides another good example. "... the *A*ll *OR*ders *S*pectral *A*lgorithm (AORSA) simulation program, developed within the *Scientific Discovery through Advanced Computing* (SciDAC) Numerical Computation of Wave Plasma-Interactions in Multidimensional Systems project, has demonstrated how electromagnetic waves can be used for driving current flow, heating, and controlling instabilities in the plasma" [6]. In the quoted article, Barrett et al. describe how a complex version of the HPL benchmark was used to double the performance of ScaLAPACK to solve a system of equations with half a million unknowns using 10,000 cores.

1.2 Original Contribution

The value of this work is in combining state-of-the-art solutions in dense linear algebra to overcome the challenges of producing a high speed implementation of the LU factorization for a heterogeneous multicore system with more than one hundred cores. Specifically, this work leverages some very recent developments, such as the parallel-recursive panel factorization [12], GPU kernel autotuning [27], and dynamic (superscalar) scheduling [18], [25]. At the same time, the solution follows the principles of the PLASMA software library, by utilizing the tile data layout and the superscalar scheduling subsystem, which makes the code ready for software integration.

1.3 Block LU Factorization

The LAPACK block LU factorization is the main point of reference here, and LAPACK naming convention is followed. The LU factorization of a matrix M has the form $M = PLU$, where L is a unit lower triangular matrix, U is an upper triangular matrix, and P is a permutation matrix. The LAPACK algorithm proceeds in the following steps: Initially, a set of nb columns (*the panel*) is factored and a pivoting pattern is produced (DGETF2). Then, the elementary transformations, resulting from the panel factorization, are applied to the remaining part of the matrix (*the trailing submatrix*). This involves swapping of up to nb rows of the trailing submatrix (DLASWP), according to the pivoting pattern, application of a triangular solve with multiple right-hand sides to the top nb rows of the trailing submatrix (DTRSM), and finally, application of matrix multiplication of the form $C = C - A \times B$ (DGEMM), where A is the panel without the top nb rows, B is the top nb rows of the trailing submatrix, and C is the trailing submatrix without the top nb rows. Then, the procedure is applied repeatedly, descending down the diagonal of the matrix.

2 Related Work

Seminal work on recursive dense linear algebra algorithms was done by Gustavson [16], who published recursive formulations of the Cholesky, LDLT, and LU factorizations. Eventually, recursion was also applied to the QR factorization by Elmroth and Gustavson [14]. Recently, Castaldo and Whaley [7] developed fast implementations of LU and QR panel operations using a technique referred to as *parallel cache assignment* (PCA). PCA builds on the *bulk synchronous parallel* model of parallelization [37], and relies on fork-and-join execution with barrier synchronizations, but allows for the preservation of data in caches in a sequence of multiple BLAS 2 operations.

Work on GPU accelerated dense linear algebra routines started before general-purpose programming environments, such as CUDA or OpenCL, were available. This time period is often referred to, somewhat ironically, as the *general purpose GPU* era. The earliest implementation of a matrix factorization was reported by Galoppo [15], who implemented the nonblocked LU decomposition without pivoting, with partial pivoting, and with full pivoting.

More papers followed when CUDA became available, largely thanks to the CUBLAS library (CUDA BLAS) provided by NVIDIA. Implementations of dense matrix factorizations were reported by Barrachina et al. [5], Baboulin et al. [3], and Castillo et al. [8]. Seminal work was done by Volkov and Demmel [38], where notably a two-GPU implementation of the LU factorization was reported, and one-dimensional block cyclic data distribution was used. It was followed by the work of Tomov et al. [35], [36] in the context of the *matrix algebra for GPUs and multicore architectures* (MAGMA) library.

An important part of these developments is the work solely focusing on optimizing matrix multiplication. Early work on tuning GEMMs in CUDA for NVIDIA GPUs targeted the previous generation of GPUs, of the GT200 architecture, such as the popular GTX 280. Pioneering work was done by Volkov and Demmel [38]. Similar efforts followed in the MAGMA project [28]. The introduction of the NVIDIA Fermi architecture triggered the development of MAGMA GEMM kernels for that architecture [30], [31], which recently evolved into a systematic autotuning approach named *automatic stencil TunerR for accelerators* (ASTRA) [27]. Other related efforts include the compiler-based work by Rudy et al. [33] and Cui et al. [10], and low-level kernel development by Nakasato [29] and Tan et al. [34].

Dense linear algebra codes, including the Cholesky, LU, and QR factorizations, have also been offloaded to the IBM Cell B. E. accelerator [9], [22], [23], [24]. Two efforts are specifically worth mentioning. Chen et al. [9] developed a single precision implementation of the LINPACK benchmark for the QS20 system, which relied on tile matrix layout and a cache-resident panel factorization. Kistler et al. [20] developed a double precision implementation of the LINPACK benchmark for the QS22 system, which employed a recursive panel factorization.

Bach et al. [4] reported an implementation of the LINPACK benchmark for a system with AMD GPUs and Deisher et al. [11] reported an implementation for the Intel MIC architecture. Both implementations follow a very different approach than the one presented in this paper.

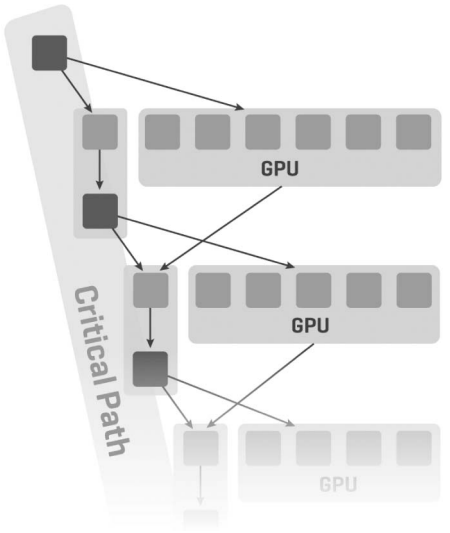

Fig. 1. The basic hybridization idea with fine-grained tasks on the critical path being dispatched to individual CPU cores and coarse-grained tasks outside of the critical path being dispatched to GPU devices.

3 Solution

The solution follows the design principles of the PLASMA numerical library by storing and processing the matrix by tiles and using dynamic, dependency-driven, runtime task scheduling. The same basic idea was previously applied to the tile QR factorization [26]. This paper builds on previous experiences to develop an implementation of a much harder algorithm in a multi-GPU scenario. The sections to follow outline the main hybridization idea, provide the motivation for the use of a tile matrix layout, describe the development of CPU and GPU kernels, explain the scheduling methodology, and discuss the communication requirements.

3.1 Hybridization

The main hybridization idea is captured in Fig. 1 and relies on representing the work as a *directed acyclic graph* (DAG) and dynamic task scheduling, with CPU cores handling the complex fine-grained tasks on the *critical path* and GPUs handling the coarse-grained data-parallel tasks outside of the critical path.

Some number of columns (*lookahead*) are assigned to the CPUs and the rest of the matrix is assigned to the GPUs in a one-dimensional block-cyclic fashion (Fig. 2). In each step of the factorization, the CPUs factor a panel and update their portion of the trailing submatrix, while the GPUs update their portions of the trailing submatrix. After each step, one column of tiles shifts from the GPUs to the CPUs (from *device memory* to *host memory*).

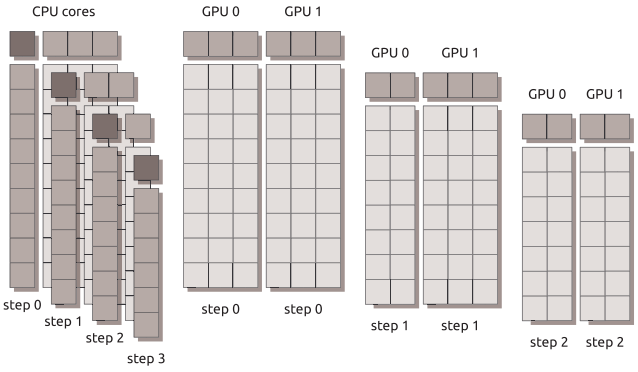

Fig. 2. The splitting of work between the CPUs and the GPUs with a number of columns on the left side (lookahead) processed by the CPUs and the remaining columns on the right side processed by the GPUs.

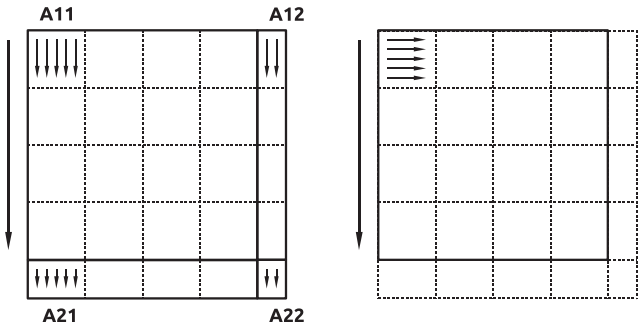

Fig. 3. Left: The CCRB layout used by the PLASMA library (Here, the A11 region defines the supported CPU layout). Right: The supported GPU layout (the A11 region of the CRRB layout).

The main advantage of this solution is the capability of overlapping the CPU processing and the GPU processing (and also overlapping of communication and computation). The GPUs have to be idle while the first panel is factored. However, the factorization of the second panel can proceed in parallel with the application of the first panel to the trailing submatrix. In practice, the level of overlapping is much bigger, i.e., the panel factorizations are a few steps ahead of updates.

3.2 Data Layout

The matrix is laid out in square tiles on the CPU side (*host memory*), where each tile occupies a continuous region of memory. Tiles are stored in column-major layout and elements within tiles are stored in column-major layout. This layout, referred to as *column-column rectangular block* (CCRB) [17], is the native layout of the PLASMA library. Here, only matrices evenly divisible into tiles are considered (Fig. 3). Inclusion in the PLASMA library would require generalization of the code to matrices that are not evenly divisible into tiles. Tiles are transposed on the GPU side (*device memory*), i.e., the layout is translated to *column-row rectangular block* (CRRB), which is critical to the performance of the row swap (DLASWP) operation (Section 3.5.1). This tilewise transposition is trivial to code and fast to execute (Section 3.5.3).

3.3 Parallel Panel on Multicore CPUs

The canonical way of performing panel factorization in the block LU algorithm is to use vector operations and matrix-vector operations (Levels 1 and 2 basic linear algebra subroutines (BLAS)). This is what the LAPACK DGETF2 routine does. Very low performance can be expected for any realistic panel sizes, due to the memory-bound nature of Level 1 and 2 BLAS. For instance, for panels of width 192 and height greater than 5,000, the DGETF2 routine barely exceeds 2 Gflop/s of performance on a typical Intel or AMD processor.

The panel factorization is in essence an LU factorization of a narrow submatrix, commonly referred to as a *tall and*

```
function xGETRFR(M, N, column) {
  if N == 1 {                                   single column, recursion stops
    idx = split_IxAMAX(...)                     compute local maximum of modulus
    gidx = combine_IxAMAX(idx)                  combine local results
    split_xSCAL(...)                            scale local data
  }
  else {
    xGETRFR(M, N/2, column)                     recursive call to factor left half
    xLASWP(...)                                 pivoting forward
    split_xTRSM(...)                            triangular solve
    split_xGEMM(...)                            Schur's complement
    xGETRFR(M, N-N/2, column+N/2)               recursive call to factor right half
    xLASWP(...)                                 pivoting backward
  }
}
```

Fig. 4. Pseudocode for the recursive panel factorization.

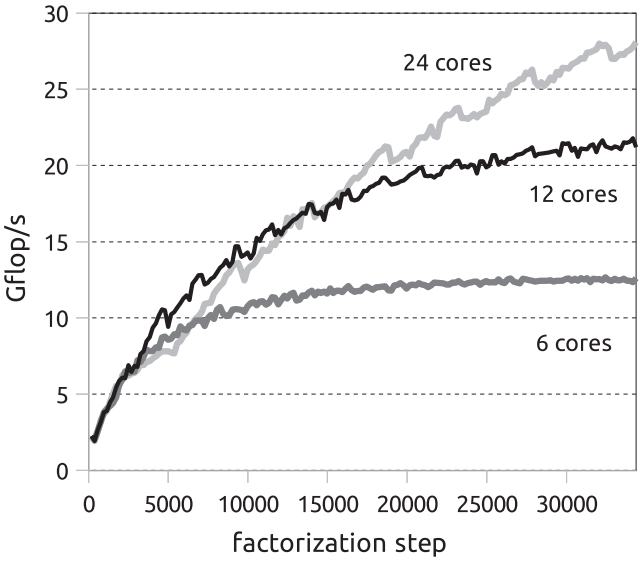

Fig. 5. Performance results for panel of width 192 with different number of cores.

skinny matrix. Therefore, it can be subdivided into a sequence of yet thinner panel factorizations and updates. For instance, the standard panel width in LAPACK is 64, so it makes sense to call the DGETRF function in LAPACK to factorize a panel of width 192. This function will internally perform three panel factorizations of width 64, by calling DGETF2, and two trailing submatrix updated, corresponding to the first two panels. Unfortunately, due to the narrow shape of the submatrices involved, this approach is only slightly more compute intensive, and the performance only goes up to 3 Gflop/s. This is still inadequate, considering that the GPUs can provide in excess of 1 Tflop/s of performance.

The problem is that it takes much longer to factor the panels than it takes to apply the corresponding updates. In such a case, the panel factorizations completely dominate the execution time, effectively nullifying the benefits of the GPUs. Clearly, much faster panel factorization is needed. A similar argument has been made for codes that do not attempt to overlap panel factorizations and updates of trailing submatrices [7].

The application of recursion allows for a decrease in memory intensity by introducing some degree of level 3 BLAS operations [16] (Fig. 4). At the same time, tiles of the panel are statically assigned to cores and each core preserves the same set of tiles throughout all the steps of the panel factorization. At some point in the LU factorization, panels become short enough to fit in the aggregate cache of the designated cores, i.e., panel operations become cache resident, which at some level resembles the technique of PCA [7] currently employed by automatically tuned linear algebra software (ATLAS). The cores are forced to work in lock step, but can benefit from a high level of cache reuse. The ultrafine granularity of operations requires very lightweight synchronization. Synchronization is implemented using *busy waiting* on volatile variables and works at the speed of hardware cache-coherency.

Fig. 5 shows the scalability of the panel implementation for panels of width 192, when using 6 cores, 12 cores (one socket), and 24 cores (two sockets). Performance of 6 cores exceeds 12 Gflop/s, performance of 12 cores exceeds 21 Gflop/s, and performance of 24 cores exceeds 28 Gflop/s. This is a tremendous performance improvement over the baseline DGETF2 and DGETRF functions.

3.4 CPU Update Kernels

The update is relatively straightforward and requires three operations: row swap (DLASWP), triangular solve (DTRSM), and matrix multiplication (DGEMM). In the case of DLASWP, one core is responsible for swaps in one column of tiles. The LAPACK DLASWP function cannot be used, because of the use of tile layout, so DLASWP with augmented address arithmetic is hand coded. In the case of DTRSM and DGEMM, one core is responsible for one tile. Calls to the Intel *Math Kernel Library* (MKL) are used with layout set to column major.

3.5 GPU Kernels

The set of required GPU kernels includes the kernels to apply the update to the trailing submatrix (DLASWP, DTRSM, and DGEMM), and the kernel to translate the panel between the CCRB layout, used on the CPU side, and the CRRB layout, used on the GPU side. The DLASWP kernel, the DTRSM kernel, and the transposition kernel are simple to write and do not have much impact on the runtime. These kernels are described first. They are followed by a longer description of the DGEMM kernel, which dominates the execution time and is the most complex.

3.5.1 DLASWP

The DLASWP routine swaps rows of the trailing submatrix according to the pivoting pattern established in the panel factorization. This operation only performs data motion, and the GPUs are very sensitive to the matrix layout in memory. In raw-major layout, threads in a warp can simultaneously access consecutive memory locations. This is not the case in column-major layout, where threads access memory with a stride. In this case, each thread generates a separate memory request, which is devastating to performance. As a result, performance is two orders of magnitude lower, and the swap operation dominates the update.

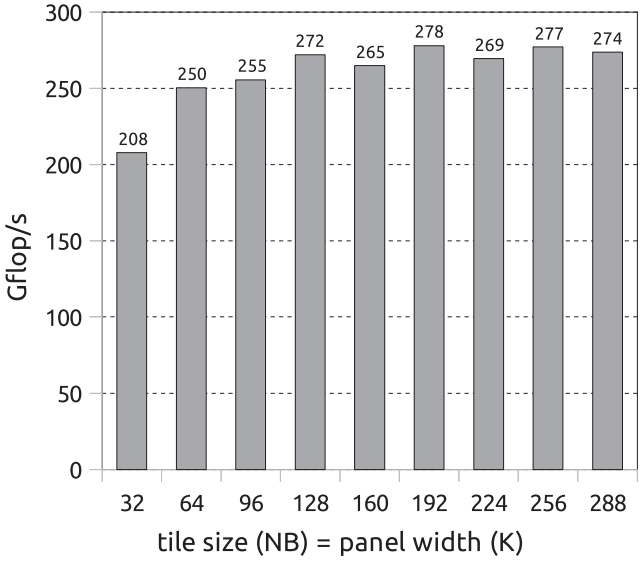

Fig. 6. Best asymptotic performance of the schur complement operation for each tiling.

This forces the use of the CRRB format, i.e., row-major storage of elements within tiles. As soon as the CRRB format is used, a straightforward implementation of the DLASWP operation completely suffices. Each thread block is tasked with swaps in one column of tiles and creates NB threads to perform them, one thread per one column of elements. This may not yet be the fastest possible way of implementing the swap. However, at this point, the impact of the swap operation on the overall performance becomes negligible.

3.5.2 DTRSM

The DTRSM routine uses the lower triangle of the $NB \times NB$ diagonal block to apply triangular solve to the block of right-hand sides formed by the top NB rows of the trailing submatrix. An efficient implementation of this routine on a GPU is difficult due to the data-parallel nature of GPUs and small size of the solve ($32 \leq NB \leq 288$).

In such a case, the standard procedure for GPUs is to replace the in-place triangular solve operation with an out-of-place multiplication of the block of right-hand sides by the inverse of the triangle. After the panel factorization, one CPU core applies the triangular solve to an $NB \times NB$ identity matrix. In the update phase, the GPUs call the DGEMM routine to apply the inverted matrix to the block of right-hand sides in an out-of-place fashion, followed by a copy of the result to the location of the original block of right-hand sides.

This operation executes at the speed of the DGEMM operation, with twice as many FLOPs as the standard DTRSM function. This is the fastest way of implementing it, known to the authors. Because it only affects a small portion of the trailing submatrix, its execution time is negligible, compared to DGEMM.

3.5.3 CCRB-CRRB Conversion

As already mentioned in Section 3.2, tile layout has numerous advantages and is the layout of choice for the PLASMA library. However, PLASMA lays out data in tiles by columns, and the GPUs require data to be laid out by rows. Otherwise, the DLASWP operation cannot perform adequately. Therefore, an operation is needed which internally transposes each tile, i.e., makes a conversion between the CCRB and the CRRB formats.

A very simple implementation is used here. Each thread block launches 1,024 threads arranged in a 32×32 grid, and each thread swaps two elements of the matrix to their transposed locations. The submatrix (column) being transposed is overlaid with a rectangular grid of blocks. Threads with the first element below the tile's diagonal perform the swap. Threads with the first element above the diagonal quit. At this point, the impact of the swap operation on the overall performance is negligible.

TABLE 1
Parameters of the Fastest Kernels

NB	$M_{blk}/N_{blk}/K_{blk}$	calc. C	read A	read B	Gflop/s
32	32×32×8	8×8	8×8	8×8	208
96	32×32×6	8×8	2×32	32×2	255
160	32×32×8	8×8	8×8	8×8	265
224	32×32×8	8×8	8×8	32×2	269
288	32×32×6	8×8	2×32	32×2	274
64	64×64×16	16×16	16×16	16×16	250
128	64×64×16	16×16	16×16	16×16	272
192	64×64×16	16×16	16×16	16×16	278
256	64×64×16	16×16	16×16	16×16	277

3.5.4 DGEMM

The DGEMM kernels are produced using the ASTRA system [27], which follows the principles of *automated empirical optimization of software*, popularized by the ATLAS [39]. The same process is currently used to produce DGEMM kernels for the MAGMA project.

The kernel is expressed through a parametrized *stencil*, creating a large search space of possible implementations. The search space is aggressively pruned, using mostly constraints related to the usage of hardware resources. On NVIDIA GPUs, one of the main selection criteria is *occupancy*, i.e., the capability of the kernel to launch a big number of *single instruction multiple threads* threads. The pruning process identifies a few tens of kernels for each tile size. The final step of autotuning is benchmarking these kernels to find the best performing ones.

There are two differences between the kernels used here and the MAGMA kernels. MAGMA kernels operate on matrices in canonical FORTRAN 77 column-major layout, compliant with the BLAS standard. The kernels used here operate on matrices in CRRB tile layout [17]. Also, MAGMA kernels are tuned for the case where all three input matrices are square, while the kernels used here are tuned for the *block outer product* operation in the LU factorization, i.e., $C = C - A \times B$, where the width of A and the height of B are equal to the matrix tile size nb.

Fig. 6 shows the performance of the fastest kernels. Table 1 also shows the corresponding values of the tuning parameters. The performance is slightly above 200 Gflop/s at $K = NB = 32$, reaches 250 Gflop/s at $K = NB = 64$ and roughly 280 Gflop/s for $K = NB = 128$, 192, and 256.

```
for (k = 0; k < SIZE; k++) {
   QUARK_Insert_Task( panel_factorization, ...          CPUs - column
   QUARK_Insert_Task( diagonal_block_inversion, ...          CPU - tile

   if (k < SIZE-1) {
      for (n = k+1; n < k+1+lookahead && n < SIZE; n++) {
         QUARK_Insert_Task( DLASWP, ...                 CPU - column
         QUARK_Insert_Task( DTRSM, ...                       CPU - tile

         for (m = k+1; m < SIZE; m++)
            QUARK_Insert_Task(DGEMM, ...                     CPU - tile
      }
   }
   if (SIZE-k-1-look > 0) {
      QUARK_Insert_Task( panel_broadcast, ...           DMA - column
      QUARK_Insert_Task( trailing_matrix_update, ...          GPUs - block
      QUARK_Insert_Task( leading_column_return, ...     DMA - column
   }
}
```

Fig. 7. Simplified QUARK code for the LU factorization.

3.6 Scheduling

Manually, multithreading the hybrid LU factorization would be nontrivial. It would be a challenge to track dependencies without automation, given the three different levels of granularity involved: single tile, one column, a large block (submatrix). Here, the QUARK superscalar scheduler [40] is used for automatic dependency tracking and work scheduling. The LU factorization code is expressed with the canonical serial loop nest (Fig. 7), where calls to CPU and GPU kernels are augmented with information about sizes of affected memory regions and directionality of arguments (IN, OUT, INOUT). QUARK schedules the work by resolving data hazards (RaW, WaR, WaW) at runtime. Three important extensions are critical to the implementation of the hybrid LU factorization: task prioritization, variable-length list of dependencies, and support for nested parallelism.

The first feature is task prioritization. It is essential that CPUs aggressively execute the critical path, i.e., traverse the DAG in a depth-first fashion. This guarantees that the panels are executed quickly and sent to the GPUs. The DAG, however, is never built in its entirety and the scheduler has no way of knowing the critical path. Instead, the critical path is indicated by the programmer, by using a priority flag when queuing the tasks in the critical path: panel factorizations and updates of the columns immediately to the right of each panel. Prioritized tasks are placed in the front of the execution queue.

The second feature is variable-length lists of parameters. CPU tasks, such as panel factorizations and row swaps, affect columns of the matrix of variable height. For such tasks, the list of dependencies is created incrementally, by looping over the tiles involved in the operation. It is a similar situation for the GPU tasks, which involve large blocks of the matrix (large arrays of tiles). The only difference is that, here, transitive (redundant) dependencies are manually removed to decrease scheduling overheads, while preserving correctness.

The third crucial extension of QUARK is support for nested parallelism, i.e., superscalar scheduling of tasks, which are internally multithreaded. The hybrid LU factorization requires parallel panel factorization for the CPUs to be able to keep pace with the GPUs. At the same time, the ultrafine granularity of the panel operations prevents the use of QUARK inside the panel. Instead, the panel is manually multithreaded using cache coherency for synchronization and scheduled by QUARK as a single task, entered at the same time by multiple threads.

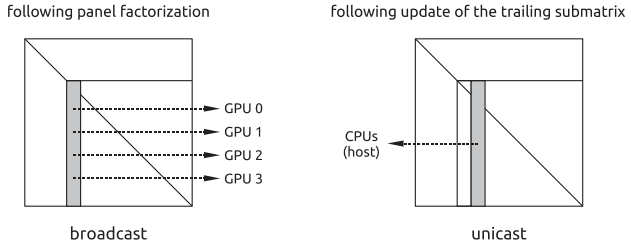

Fig. 8. CPU-GPU (host to device) communication.

3.7 Communication

Communication is shown on Fig. 8. Each panel factorization is followed by a broadcast of the panel to all the GPUs. After each update, the GPU in possession of the leading leftmost column sends that column back to the CPUs (host memory). These communications are expressed as QUARK tasks with proper dependencies linking them to the computational tasks. Because of the use of lookahead, the panel factorizations can proceed ahead of the trailing submatrix updates and so can transfers, which allows for perfect overlapping of communication and computation, as further discussed in the following section.

4 Results

This section includes a precise description of the hardware-software environment, followed by the performance results and a detailed discussion.

4.1 Hardware and Software

The system used for this work couples one CPU board with four sockets and one GPU board with four sockets. The CPU board is an NVIDIA Tesla S2050 system with four Fermi chips, 14 multiprocessors each, clocked at 1.147 GHz. The CPU board is a H8QG6 Supermicro system with 4 AMD Magny Cours chips, 12 cores each, clocked at 2.1 GHz.

The theoretical peak of a single CPU socket amounts to $2.1\ \text{GHz} \times 12\ \text{cores} \times 4\ \text{ops per cycle} \simeq 101\ \text{Gflop/s}$, making it ~ 403 Gflop/s for all four CPU sockets. The theoretical peak of a single GPU amounts to $1.147\ \text{GHz} \times 14\ \text{cores} \times 32\ \text{ops per cycle} \simeq 514\ \text{Gflop/s}$, making it $\sim 2,055$ Gflop/s for all four GPUs. The combined CPU-GPU peak is ~ 2459 Gflop/s.

The system runs Linux kernel version 2.6.35.7 (Red Hat distribution 4.1.2-48). The CPU part of the code is built using GCC 4.4.4. Intel MKL version 2011.2.137 is used for BLAS calls on the CPUs. The GPU part of the code is built using CUDA 4.0.

4.2 Performance

Fig. 9 shows the overall performance of the hybrid LU factorization, and Table 2 lists the exact performance number for each point along with values of tuning parameters. Tuning is done by exhaustive search across all parameters. Matrix size goes up to 34,560. Beyond that

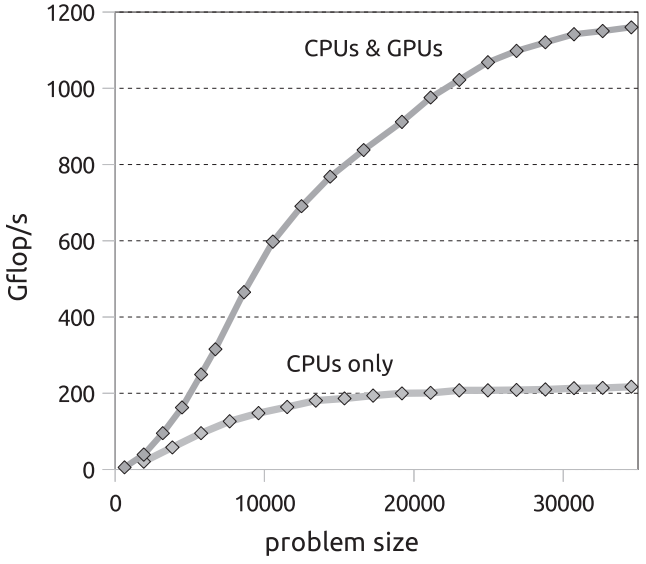

Fig. 9. Overall performance of the LU factorization.

point, the size of memory on all GPUs is exceeded. Each GPU can provide 2.6 GB of *error correcting code* protected memory. For comparison, the light gray line shows the performance of a CPU-only run using all 48 CPU cores, which is equivalent in behavior and performance to a call to the DGETRF routine in PLASMA.

Fig. 10 shows a small fragment in the middle of a 23,040 run (the smallest size exceeding 1 Tflop/s performance). In the CPU part, only the panel factorizations are shown. The steps shown on the figure correspond to factoring submatrices of size $\sim 12{,}000$. Due to the deep lookahead, panel factorizations on the CPUs run a few steps ahead of trailing submatrix updates on the GPUs. This allows for perfect overlapping of CPU work and GPU work. It also allows for perfect overlapping of communication between the CPUs and the GPUs, i.e., between the host memory and the device memories. Each panel factorization is followed by a broadcast of the panel to the GPUs (light gray DMA). Each trailing submatrix update is followed by returning one column to the CPUs (dark gray DMA).

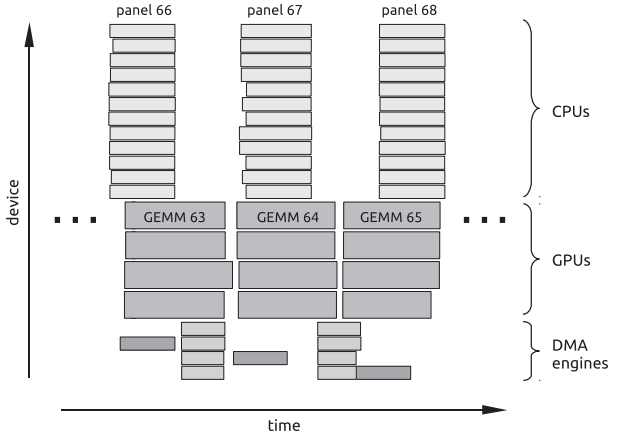

Fig. 10. A small portion in the middle of the 1 Tflop/s run.

Fig. 11 shows the performance of the panel factorization throughout the largest run (34,560), using different numbers of cores, for panels of width 192. The jagged shape of the lines reflects the fact that the panel cores have to compete for main memory with the other cores, applying updates at the same time. Generally, more cores provide higher performance, due to more computing power and larger capacity of their combined caches. However, 24 cores (two sockets) provide only a small performance improvement over 12 cores (single socket) due to the higher cost of intersocket communication over communication within the same socket. In actual LU runs, the use of 12 cores turns out to always be optimal, even for large matrices. While 12-core panel factorizations are capable of keeping up with GPU updates, the remaining cores can be committed to CPU updates.

Fig. 12 shows the performance of the GPU DGEMM kernel throughout the entire factorization. The gray line shows the DGEMM kernel performance on a single GPU. The black line shows the performance of the 4-GPU DGEMM task. The jagged shape of the line is due to the

TABLE 2
LU Performance and Values of Tuning Parameters

size	NB	lookahead	panel cores	Gflop/s
640	64	1	12	6
1,920	″	″	″	39
3,200	″	″	″	95
4,480	″	″	″	163
5,760	″	″	″	249
6,720	96	″	″	315
8,640	″	2	″	465
10,560	″	″	″	598
12,480	″	″	″	690
14,400	″	3	″	768
16,640	128	5	″	838
19,200	192	12	″	912
21,120	″	″	″	976
23,040	″	″	″	1022
24,960	″	″	″	1068
26,880	″	″	″	1098
28,800	″	13	″	1121
30,720	″	14	″	1142
32,640	″	″	″	1150
34,560	″	″	″	1160

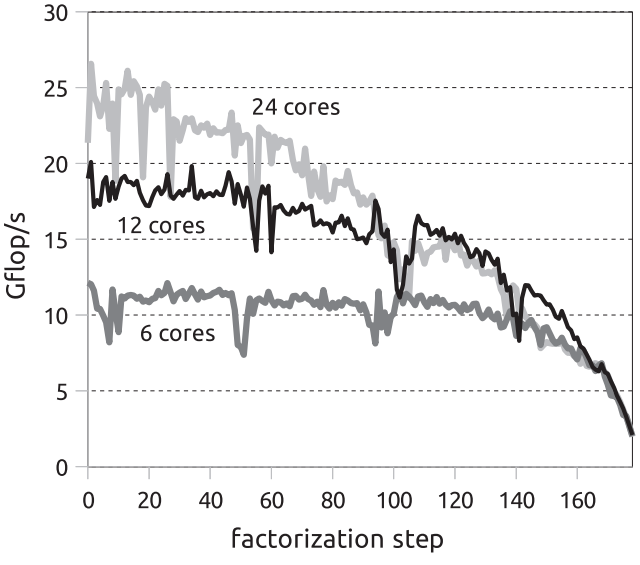

Fig. 11. Performance of panel factorization in each step of matrix factorization.

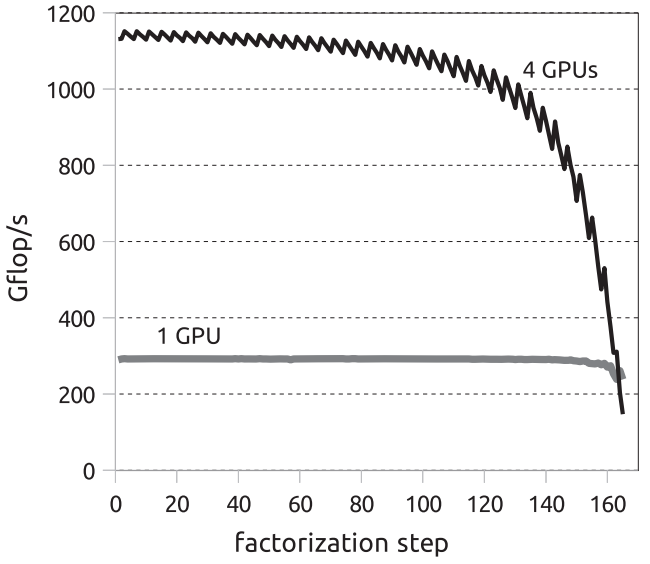

Fig. 12. Performance of DGEMM in each step of matrix factorization.

load imbalance among the GPUs. The high peaks correspond to the calls where the load is perfectly balanced, i.e., the number of columns updated by the GPUs is divisible by 4. When this is not the case, the number of columns assigned to different GPUs can differ by one. The load imbalance can be completely eliminated by scheduling the GPUs independently. Although, potential performance benefits are on the order of a few percent.

5 Conclusions

The results reveal the challenges of programming a hybrid multicore system with accelerators. There is a disparity in the performance of the CPUs and the GPUs to start with. It turns into a massive disproportion when the CPUs are given the difficult (synchronization-rich and memory-bound) task of panel factorization, and the GPUs are given the easy (data-parallel and compute-bound) task of matrix multiplication. While the performance of panel factorization on the CPUs is roughly at the level of 20 Gflop/s, the performance of matrix multiplication on the GPUs is almost at the level of 1,200 Gflop/s (two orders of magnitude). The same disproportion applies to the computational power of the GPUs versus the communication bandwidth between the CPU memory and the GPU memory (host to device). The key to achieving good performance under such adverse conditions is overlapping of CPU processing and GPU processing, and overlapping of communication.

6 Software

The code is available from the authors upon request. If released, the code will be available under the modified BSD license.

Acknowledgments

The authors thank David Luebke, Steven Parker, and Massimiliano Fatica for their insightful comments about the Fermi architecture.

References

[1] 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, http://www.netlib.org/lapack/lug/. 1992.

[2] D.B. Kirk and W.W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach,* Applications of GPU Computing Series. Morgan Kaufmann, 2010.

[3] M. Baboulin, J.J. Dongarra, and S. Tomov, "LAPACK Working Note 200: Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures," Technical Report UT-CS-08-615, Electrical Eng. and Computer Science Dept., Univ. of Tennessee, www.netlib.org/lapack/lawnspdf/lawn200.pdf" www.netlib.org/lapack/lawnspdf/lawn200.pdf, 2008.

[4] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr, "Optimized HPL for AMD GPU and Multi-Core CPU Usage," *Computer Science: Research Development,* vol. 26, nos. 3/4, pp. 153-164, 2011, DOI: 10.1007/s00450-011-0161-5.

[5] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and E.S. Quintana-Orti, "Solving Dense Linear Systems on Graphics Processors," *Proc. 14th Int'l Euro-Par Conf. Parallel Processing,* pp. 739-748, Aug. 2008, DOI: 10.1007/978-3-540-85451-7_79.

[6] R.F. Barrett, T.H.F. Chan, E.F. D'Azevedo, E.F. Jaeger, K. Wong, and R.Y. Wong, "Complex Version of High Performance Computing LINPACK Benchmark (HPL)," *Concurrency Computation: Practice Experience,* vol. 22, no. 5, pp. 573-587, 2009, DOI: 10.1002/cpe.1476.

[7] A.M. Castaldo and R.C. Whaley, "Scaling LAPACK Panel Operations Using Parallel Cache Assignment," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '10),* Jan. 2010, DOI: 10.1145/1693453.1693484.

[8] M. Castillo, E. Chan, F.D. Igual, R. Mayo, E.S. Quintana-Orti, G. Quintana-Orti, R. van de Geijn, and F.G. Van Zee, "FLAME Working Note 31: Making Programming Synonymous with Programming for Linear Algebra Libraries," Technical Report TR-08-20, Computer Science Dept., Univ. of Texas at Austin, www.cs.utexas.edu/users/flame/pubs/flawn31.pdf, 2008.

[9] T. Chen, R. Raghavan, J.N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and Its First Implementation—A Performance View," *IBM J. Research & Development,* vol. 51, no. 5, pp. 559-572, 2007, DOI: 10.1147/rd.515.0559.

[10] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng, "Automatic Library Generation for BLAS3 on GPUs," *Proc. Int'l Parallel and Distributed Processing Symp.,* May 2011, DOI: 10.1109/IPDPS.2011.33.

[11] M. Deisher, M. Smelyanskiy, B. Nickerson, V.W. Lee, M. Chuvelev, and P. Dubey, "Designing and Dynamically Load Balancing Hybrid LU for Multi/Many-Core," *Computer Science Research and Development,* vol. 26, no. 3/4, pp. 211-220, 2011, DOI: 10.1007/s00450-011-0169-x.

[12] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, "LAPACK Working Note 259: Achieving Numerical Accuracy and High Performance Using Recursive Tile LU Factorization," Technical Report UT-CS-11-688, Electrical Eng. and Computer Science Dept., Univ. of Tennessee, http://www.netlib.org/lapack/lawnspdf/lawn259.pdf, 2011.

[13] J.J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, Present and Future," *Concurrency Computation: Practice Experience,* vol. 15, no. 9, pp. 803-820, 2003, DOI: 10.1002/cpe.728.

[14] E. Elmroth and F.G. Gustavson, "Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance," *IBM J. Research and Development,* vol. 44, no. 4, pp. 605-624, 2000, DOI: 10.1147/rd.444.0605.

[15] N. Galoppo, N.K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware," *Proc. ACM/IEEE Conf. Supercomputing,* Nov. 2005, DOI: 10.1109/SC.2005.42.

[16] F.G. Gustavson, "Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms," *IBM J. Research Development,* vol. 41, no. 6, pp. 737-756, 1997, DOI: 10.1147/rd.416.0737.

[17] F.G. Gustavson, L. Karlsson, and B. Kågström, "Parallel and Cache-Efficient in-Place Matrix Storage Format Conversion." *ACM Trans. Math. Software,* vol. 38, no. 3, article no. 17, 2012, DOI: 10.1145/2168773.2168775.

[18] A. Haidar, H. Ltaief, A. YarKhan, and J.J. Dongarra, "Analysis of Dynamically Scheduled Tile Algorithms for Dense Linear Algebra on Multicore Architectures," *Concurrency Computation: Practice Experience,* vol. 24, pp. 305-321, 2011, DOI: 10.1002/cpe.1829.

[19] *GPU Computing Gems Jade Edition,* Applications of GPU Computing Series, W.W. Hwu, ed. Morgan Kaufmann, 2011.

[20] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Programming the Linpack Benchmark for the IBM PowerXCell 8i Processor," *Scientific Programming,* vol. 17, nos. 1/2, pp. 43-57, 2009, DOI: 10.3233/SPR-2009-0278.

[21] *Scientific Computing with Multicore and Accelerators,* J. Kurzak, D.A. Bader, J. Dongarra, eds. Chapman & Hall, 2010.

[22] J. Kurzak, A. Buttari, and J.J. Dongarra, "Solving Systems of Linear Equation on the CELL Processor Using Cholesky Factorization," *Trans. Parallel Distributed System,* vol. 19, no. 9, pp. 1175-1186, 2008, DOI: TPDS.2007.70813.

[23] J. Kurzak and J.J. Dongarra, "Implementation of Mixed Precision in Solving Systems of Linear Equations on the CELL Processor," *Concurrency Computation: Practice Experience,* vol. 19, no. 10, pp. 1371-1385, 2007, DOI: 10.1002/cpe.1164.

[24] J. Kurzak and J.J. Dongarra, "QR Factorization for the Cell Broadband Engine," *Scientific Programming,* vol. 17, nos. 1/2, pp. 31-42, 2009, DOI: 10.3233/SPR-2009-0268.

[25] J. Kurzak, H. Ltaief, J.J. Dongarra, and R.M. Badia, "Scheduling Dense Linear Algebra Operations on Multicore Processors," *Concurrency Computation: Practice Experience,* vol. 21, no. 1, pp. 15-44, 2009, DOI: 10.1002/cpe.1467.

[26] J. Kurzak, R. Nath, P. Du, and J.J. Dongarra, "An Implementation of the Tile QR Factorization for a GPU and Multiple CPUs," *Proc. State of the Art in Scientific and Parallel Computing Conf.,* pp. 248-257, June 2010, DOI: 10.1007/978-3-642-28145-7.

[27] J. Kurzak, S. Tomov, and J. Dongarra, "LAPACK Working Note 245: Autotuning GEMMs for Fermi," Technical Report UT-CS-11-671, Electrical Eng. and Computer Science Dept., Univ. of Tennessee, www.netlib.org/lapack/lawnspdf/lawn245.pdf, 2011.

[28] Y. Li, J. Dongarra, and S. Tomov, "A Note on Auto-Tuning GEMM for GPUs," *Proc. Int'l Conf. Computational Science,* pp. 884-892, May 2009, DOI: 10.1007/978-3-64 2-01970-8_89.

[29] N. Nakasato, "A Fast GEMM Implementation on a Cypress GPU," *Proc. First Int'l Workshop Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems,* http://www.dcs.warwick.ac.uk/sdh/pmbs10/pmbs10/Workshop_Programme_file s/fastgemm.pdf, Nov. 2010.

[30] R. Nath, S. Tomov, and J. Dongarra, "Accelerating GPU Kernels for Dense Linear Algebra," *Proc. Int'l Meeting High Performance Computing for Computational Science,* pp. 83-92, June 2010, DOI: 10.1007/978-3-642-19328-6_10.

[31] R. Nath, S. Tomov, and J. Dongarra, "An Improved MAGMA GEMM for Fermi Graphics Processing Units," *Int'l J. High Performance Computing Applications,* vol. 24, no. 4, pp. 511-515, 2010, DOI: 10.1177/1094342010385 729.

[32] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum,* vol. 26, no. 1, pp. 80-113, 2007, DOI: 10.1111/j.146 7-8659.2007.01012.x.

[33] G. Rudy, M.M. Khan, M. Hall, C. Chen, and J. Chame, "A Programming Language Interface to Describe Transformations and Code Generation," *Proc. 23rd Int'l Workshop Languages and Compilers for Parallel Computing,* pp. 136-150, Oct. 2010, DOI: 10.1007/978-3-642-19595-2_10.

[34] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast Implementation of DGEMM on Fermi GPU," *Proc. IEEE/ACM Supercomputing Conf.,* Nov. 2011, DOI: 10.1145/2063384.2063431.

[35] S. Tomov, J. Dongarra, and M. Baboulin, "Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems," *Parallel Computing,* vol. 36, nos. 5/6, pp. 232-240, 2010, DOI: 10.1016/j.parco.20 09.12.005.

[36] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense Linear Algebra Solvers for Multicore with GPU Accelerators." *Proc. IEEE Int'l Parallel and Distributed Processing Symp.,* pp. 1-8, Apr. 2010, DOI: 10.1109/IPDPSW.201 0.5470941.

[37] L.G. Valiant, "A Bridging Model for Parallel Computation." *Comm. ACM,* vol. 33, no. 8, pp. 103-111, 1990, DOI: 10.1145/79173.79181.

[38] V. Volkov and J.W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," *Proc. ACM/IEEE Conf. Supercomputing,* Nov. 2008, DOI: 10.1145/1413370.1413402.

[39] R.C. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing,* vol. 27, nos. 1-20, pp. 3-35, 2001, DOI: 10.1016/S0167-81 91(00)00087-9.

[40] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: Queueing and Runtime for Kernels," Technical Report ICL-UT-11-02, Innovative Computing Laboratory, Univ. of Tennessee, http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf, Apr. 2011.

Jakub Kurzak received the MSc degree in electrical and computer engineering from the Wroclaw University of Technology, Poland, and the PhD degree in computer science from the University of Houston. He is a research director at the Innovative Computing Laboratory in the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville. His research interests include parallel algorithms, specifically in the area of numerical linear algebra, and also parallel programming models and performance optimization for parallel architectures, multicore processors, and GPU accelerators. He is a member of the IEEE.

Piotr Luszczek is a research director at the University of Tennessee Knoxville's Innovative Computing Laboratory. His core research activity is centered around performance modeling and evaluation. He has extensive experience with high performance numerical linear algebra and signal processing codes that achieve high efficiency on a varied array of hardware architectures, including massively parallel high end distributed memory machines, shared memory servers, and mobile platforms that all feature specialized and general purpose accelerators running on the major operating systems. His research also revolves around long-term energy consumption and performance trends in high performance and cloud computing. His contributions to the scientific community include conference proceedings, journals, book chapters, and patent applications that showcase his main research agenda and expertize, as well as programming paradigms, parallel language design and productivity aspects of high performance scientific computing. He is a member of the IEEE.

Mathieu Faverge received the PhD degree in computer science from the University of Bordeaux 1, France. He is a postdoctoral research associate at the University of Tennessee Knoxville's Innovative Computing Laboratory. His main research interests are numerical linear algebra algorithms for sparse and dense problems on massively parallel architectures, and especially DAG algorithms relying on dynamic schedulers. He has experience with hierarchical shared memory, heterogeneous and distributed systems, and his contributions to the scientific community include efficient linear algebra algorithms for those systems. He is a member of the IEEE.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He received the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches; in 2008, he received the first IEEE Medal of Excellence in Scalable Computing; in 2010, he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; and in 2011, he received the IEEE IPDPS 2011 Charles Babbage Award. He is a fellow of the AAAS, ACM, and SIAM, and a member of the National Academy of Engineering. He is a life fellow of the IEEE.