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 article 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 systems lies in the disproportion between the computatinal power of the CPUs, compared to the GPUs, and in the meager bandwidth of the communication link between their memory systems. 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 towards 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. Since the algorithm is coded in double precision, it can serve as the basis for an implementation of the High Performance Linpack benchmark (HPL) [14]. 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* (SM). 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 Non-Uniform Memory Access* (ccNUMA) shared memory system. The GPUs have their private memories, referred to at *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* (PCIe) 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 schedulig,
- communication and computation overlapping.

Notably, the level of performance reported in this work could be accomplished thanks to recently developed capatilities, 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 signle 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 [4], [19], [22], [34].

1.2 Original Contribution

The main original contribution of this paper is in developing highly optimized CPU and GPU components of the LU algorithm and then uniquely combining them throught the use of a dynamic scheduler and the technique of lookahead. This

- *J. Kurzak, P. Luszczek, M. Faverge and J. Dongarra are with the Electrical Engineering and Computer Science Department, University of Tennessee.*
- *J. Dongarra is also with Computer Science and Mathematics Division, Oak Ridge National Laboratory and School of Mathematics and School of Computer Science, University of Manchester.*

work leverages recent developments in parallel panel factorizations [13], GPU kernel autotuning [28] and dynamic (superscalar) scheduling of dense linear algebra operations [18], [26]. The authors are not aware of any other multi-CPU, multi-GPU implementation of the LU factorization capable of reaching similar level of performance for similar range of problem sizes.

2 Background

The following three sections provide a brief introduction to the vital concepts in this work: the block LU factorization algorithm, the superscalar scheduling methodology and the NVIDIA CUDA programming system, followed by the summary of closely related previous developments in this ares.

2.1 Block LU Factorization

The LU factorization of a matrix A has the form

$$A = PLU,$$

where L is a unit lower triangular matrix, U is an upper triangular matrix and P is a permutation matrix. The block LU factorization algorithm [12] proceeds in the following steps: Initially, a set of NB columns (*the panel*) is factored and a pivoting pattern is produced. Then the elementary transformations, resulting from the panel factorization, are applied in block fashion to the remaining part of the matrix (*the trailing submatrix*). First, NB rows of the trailing submatrix are swapped, according to the pivoting pattern. Then a triangular solve is applied to the top NB rows of the trailing submatrix. Finally, matrix multiplication of the form $A_{ij} \leftarrow A_{ij} - A_{ik} \times A_{kj}$ is performed, where A_{ik} is the panel without the top NB rows, A_{kj} is the top NB rows of the trailing submatrix and A_{ij} is the trailing submatrix without the top NB rows. Then the procedure is applied repeatedly, descending down the diagonal of the matrix (Figure 1). The block algorithm is described in detail in section 2.6.3. of the book by Demmel [12]

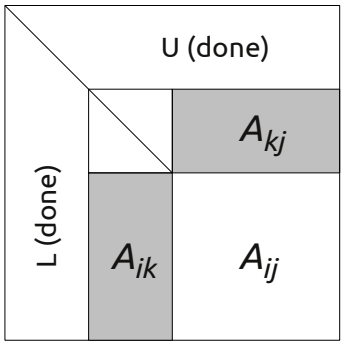

Fig. 1. Block LU factorization (Level 3 BLAS algorithm of LAPACK) [12].

In the LAPACK software library, panel factorization is performed by the `DGETF2` routine, swapping of rows by the `DLASWP` routine, triangular solve by the `DTRSM` routine and matrix multiplication by the `DGEMM` routine. The `DGETF2` and `DLASWP` routines are implemented in LAPACK, while the `DTRSM` and `DGEMM` routines are part of the *Basic Linear Algebra Subroutines* (BLAS) standard. LAPACK is an academic project and therefore the source code is freely distributed online. BLAS is a set of standardized routines, and is available in commercial packages (e.g. MKL from Intel, ACML from AMD, ESSL from IBM), in academic packages (e.g. ATLAS) and also as a reference implementation in FORTRAN 77 from the Netlib software repository. Notably, the last source provides an unoptimized code, which is only meant to serve as the definition of BLAS.

2.2 Superscalar Scheduling

Superscalar schedulers exploit multithreaded parallelism in a similar way as sperscalar processors exploit *Instruction Level Parallelism* (ILP). Scheduling proceeds under the constraints of data hazards: *Read after Write* (RaW), *Write after Read* (RaW) and *Write after Write* (RaW). In addition, the WaR and WaW dependencies can be removed by using the tachnique of renaming. The first processor archtecture to use superscalar scheduling was the CDC 6600, and the first architecture to use register renaming was the IBM System/360 (Tomasulo algorithm).

In the context of multithreading, superscalar scheduling is a way of automatically parallelizing serial code. The programmer is responsible for encapsulating the work in side-effect-free functions (parallel tasks) and providing directionality of their parameters (input, output, inout), and the scheduling is left to the runtime. Scheduling is done by conceptually exploring the *Directed Acyclic Graph* (DAG), or task graph, of the problem. In practice the DAG is explored in a sliding window fashion and actually not explicitly built. It is represented inplicitly through data structures such as linked lists and hash tables, keeping track of data dependencies.

The oldest system, that the authors are aware of, is the Jade project from Stanford Univesity [37], [38]. Two well established projects are StarSs from Barcelona Supercomputer Center [35], [36] and StarPU from INRIA Bordeaux [3]. The scheduler used in this work is *QUeuing And Runtime for Kernels* (QUARK) [46], currently the system of choice for the PLASMA project [20]. It is suitable for this work due to a number of vital extensions, discussed in more detail in section 3.6.

2.3 NVIDIA CUDA

In November 2006, NVIDIA introduced the *Compute Unified Device Architecture* (CUDA™), a general purpose parallel computing architecture, with a new parallel programming model and instruction set architecture, that leverages the parallel compute engine in NVIDIA GPUs to solve complex computational problems [33].

At its core are three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization, that are exposed to the programmer as a set of language extensions. They guide the programmer to partition the problem

into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads.

The CUDA architecture is built around a scalable array of multithreaded *Streaming Multiprocessors* (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called *Single-Instruction, Multiple-Thread* (SIMT). The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively with simultaneous hardware multithreading. However, unlike CPU cores, they are issued in order and there is no branch prediction and no speculative execution.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term warp originates from weaving, the first parallel thread technology. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution.

2.4 Related Work

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

More papers followed when CUDA became available, largly thanks to the CUBLAS library (CUDA BLAS) provided by NVIDIA. Implementations of dense matrix factorizations were reported by Barrachina et al. [6], Baboulin et al. [5], and Castillo et al. [9]. Seminal work was done by Volkov and Demmel [45], where notably a two-GPU implementation of the LU factorization was reported, and 1D block cyclic data distribution was used. It was followed by the work of Tomov et al. [42], [43] in the context of the *Matrix Algebra for GPUs and Multicore Architectures* (MAGMA) library.

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 [45]. Similar efforts followed in the MAGMA project [29]. The introduction of the NVIDIA Fermi architecture triggered the development of MAGMA GEMM kernels for that architecture [31], [32], which recently evolved into a systematic autotuning approach named *Automatic Stencil TunerR for Accelerators* (ASTRA) [28]. Other related efforts include the compiler-based work by Rudy et al. [39] and Cui et al. [11], and low-level kernel development by Nakasato [30] and Tan et al. [41].

Dense linear albebra codes, including the Cholesky, LU and QR factorizations have also been offloaded to the IBM Cell B. E. accelerator [10], [23]–[25]. Two efforts are specifically worth mentioning. Chen et al. developed a single precision implementation of the Linpack benchmark for the QS20 system, which relied on a tile matrix layout and a cache-resident panel factorization [10]. Kistler et al. developed a double precision implementation of the Linpack benchmark for the QS22 system, which employed a recursive panel factorization [21].

Panel factorization has been successfully parallelized and incorporated into a general LU factorization code [7] using an optimized implementation of mostly Level 1 BLAS. This was done in a flat parallelism model with *Block Synchronous Processing* (BSP) model [44] also referred to as fork-join execution. The authors refer to their approach as Parallel Cache Assignment (PCA). Our work on parallelizing the panel factorization [13] differs in a few key aspects. We employ recursive formulation of the factorization [16] and therefore are able to use Level 3 BLAS as opposed to just Level 1 BLAS. Another important difference is the nested parallelism with which we have the flexibility to allocate only a small set of cores for the panel work while other cores carry on with the remaining tasks such as the Schur complement updates. Finally, we use dynamic scheduling that executes fine grained tasks asynchronously, which is drastically different from a BSP or fork-join parallelism.

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 approach was successfully applied to the QR factorization in the past [27]. Here similar methodology is applied to the LU factorization for a system with multiple GPUs. 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 on Figure 2 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.

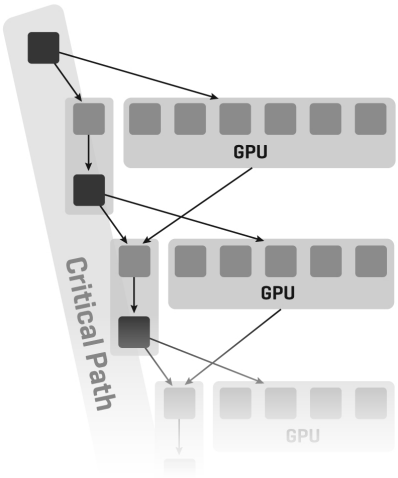

Fig. 2. The basic hybridization idea, with fine-grained tasks on the critical path being dispatched to individual CPU cores and coarse-grained tasks outside of the critical path being dispatched to GPU devices.

Some number of columns (*lookahead*) are assigned to the CPUs and the rest of the matrix is assigned to the GPUs in a 1D block-cyclic fashion (Figure 3). 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*).

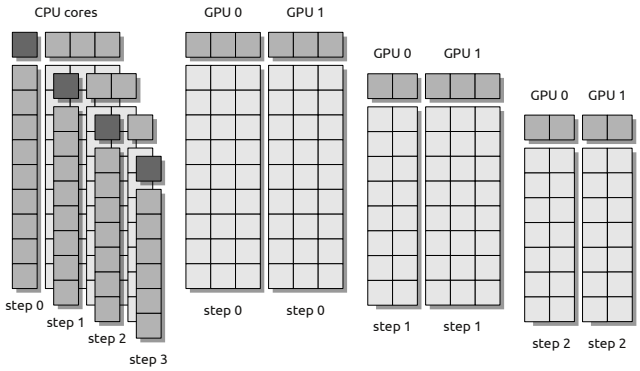

Fig. 3. The splitting of work between the CPUs and the GPUs, with a number of columns on the left side (lookahead) processed by the CPUs and the remaining columns on the right side processed by the GPUs.

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 prallel 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, where each tile occupies a continuous region of memory. Tiles are stored in column-major. Elements within tiles are stored in row-major. Such layout is referred to as *Column-Row Rectangular Block* (CRRB) [17] Here, only matrices evenly divisible into tiles are considered (Figure 4). This layout is preserved on the CPU side (*host memory*) and the GPU side (*device memory*). The storage of elements by rows is critical to the performance of the row swap (`DLASWP`) operation on the GPUs. The storage of tiles by columns simplifies the communication of columns between the CPUs and the GPUs.

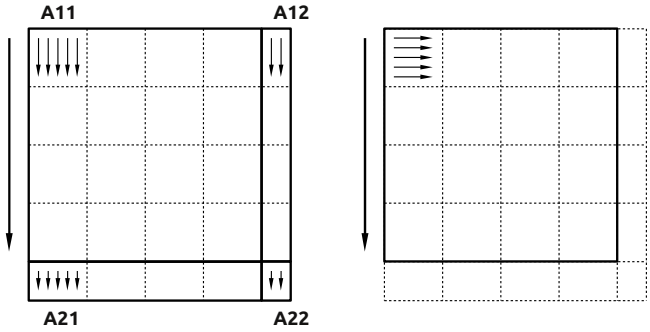

Fig. 4. Left: the CCRB layout used in the PLASMA library; Right: the CRRB layout used here.

The CRRB layout seems to be the cleanest solution for a stand-alone prototype described in this work. For integration into the PLASMA library, the CCRB layout can be used on the CPU side and the CRRB layout on the GPU side, with a translation stage in between. The translations would be associated with communication and only apply to one column of tiles at a time. The fact that both layouts are tile layouts, greatly simplify the translation, which can be efficiently applied on the GPU side, without much impact on the overall performance. Inclusion in the PLASMA library also requires generalization of the code to matrices that are not evenly divisible into tiles.

3.3 Parallel Panel on Multicore CPUs

Panel factorization in LU implementations has most commonly been performed with a sequential routined called xGETF2() that calls BLAS Level 1 and 2. On our tested machine this routine barely exceeds 2 Gflop/s on panels taller than 5000 rows and 192 columns wide. Calling xGETRF() increases the performance by 1 Gflop/s. With availability of 1 Tflop/s of combined performance coming from the hardware accelerators it takes twice as long (about 0.35

```
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. 5. Pseudo-code for the recursive panel factorization.

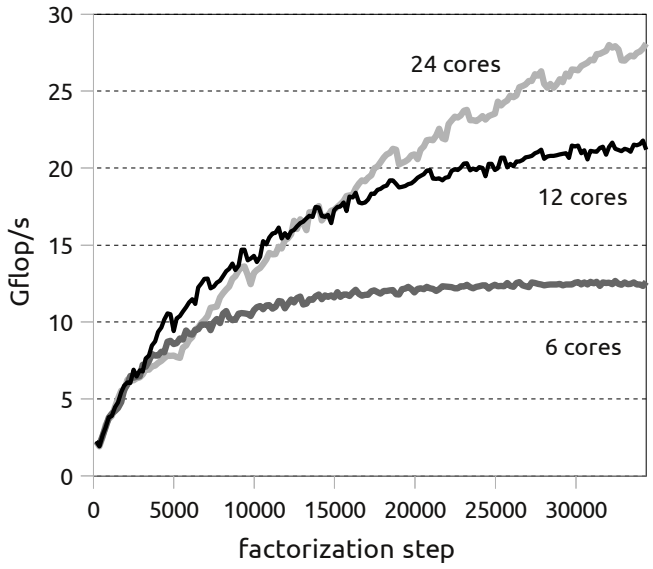

Fig. 6. Performance results for panel of width 192.

seconds) to factor the first panel than it takes to apply Schur complement update from that panel (about 0.15 seconds). With such performance imbalance it is no longer possible to hide the time for the factorization of the panel behind the time for taken by other tasks. It is clear that we needed a much faster panel factorization. A similar argument has been made in codes that do not use hiding of the panel factorization [8].

We started with the recursive formulation of LU factorization [16] for our implementation. Figure 5 shows the pseudo-code of our implementation. Aside from gaining high level formulation free of low level tuning parameters, recursive formulation affords us to dispense of a higher level tuning parameter commonly called algorithmic blocking. There is already panel width – a tunable value used for merging multiple panel columns together. Non-recursive panel factorizations could potentially establish another level of tuning called *inner-blocking* [1], [2]. This is avoided in our implementation.

From the hardware architecture perspective, our panel factorization relies on the combined size of caches – insufficient amount of such combined cache results in poor performance for very tall panels as shown in Figure 6 for a run on 6 cores.

The challenging part of the parallelization is the fact that the recursive formulation suffers from inherent sequential control flow that is characteristic of the column-oriented implementation employed by LAPACK and ScaLAPACK. As a first step then, we apply a 1D partitioning technique that has proven successful before [8]. We employed this technique for the recursion-stopping case: single column factorization. Due to the fact that our storate is CCRB layout we need to only slightly modify our code to account for iteration over the matrix tiles.

We consider data partitioning among the threads to be of paramount importance for our code and its succesful parallelization. Unlike the PCA method [8], we do not perform extra data copy to eliminate memory effects that are detrimental to performance such as TLB misses, false sharing, etc. By choosing the recursive formulation, we rely instead on Level 3 BLAS to perform these optimizations for us. Not surprisingly, this was also the goal of the original recursive algorithm and its sequential implementation [16]. What is left to do is the introduction of parallelism that is commonly missing from Level 3 BLAS when narrow rectangular matrices are involved. Namely, the use of Level 3 BLAS call for triangular solve – xTRSM() and LAPACK's auxiliary routine for swapping named xLASWP(). Both of these calls do not readily lend themselves to our CCRB 1D partitioning scheme due to two main reasons: (1) each call to these functions occurs with a variable matrix size, and (2) 1D partitioning makes the calls dependent upon each other thus creating synchronization overhead. The latter problem is fairly easy to see as the pivoting requires data accesses across the entire column and memory locations may be considered random. Each pivot element swap would then require coordination between the threads that the column is partitioned amongst. The former issue is more subtle in that the overlapping regions of the matrix create a memory hazard that may be at times masked by the synchronization effects occurring in other portions of the factorization. To deal with both issues at once, we chose to use 1D partitioning across the columns and not across the rows as before. This removes the need for extra synchronization and affords us parallel execution, albeit to a limited extent due to the narrow size of the panel.

The Schur's complement update is commonly implemented by a call to Level 3 BLAS kernel xGEMM() and this is also a new function that is not present within the panel factorizations from LAPACK and ScaLAPACK. Parallelizing this call is much easier than all the other new components of our panel factorization. We chose to reuse the across-columns 1D partitioning to simplify the management of overlapping memory references and to again reduce resulting synchronization points.

Instead of low level memory optimizations, we turned our focus towards avoiding synchronization points and let the computation proceed asynchronously and independently as long as possible until it is absolutely necessary to perform

communication between threads. One design decision that stands out in this respect is the fixed partitioning scheme. Regardless of the current column height (within the panel being factored), we always assign the same amount of rows to each thread except for the first thread. This causes a load imbalance as the thread number 0 has progressively smaller amounts of work to perform as the panel factorization progresses from the first to the last column. This is counterbalanced by the fact that the panels are relatively tall compared to the number of threads and the first thread usually has greater responsibility in handling pivot bookkeeping and synchronization duties.

Figure 6 shows a scalability study on the tested machine of our parallel recursive panel LU factorization with width of 192. We limit our parallelism level to 24 cores because our main factorization needs the remaining cores for trailing matrix updates. When compared with the panel factorization routine xGETF2() (that uses only BLAS Level 1 and 2), we achieve super-linear speedup for a wide range of panel heights with the maximum achieved efficiency exceeding 550%. In an arguably more relevant comparison against the xGETRF() routine, which could be implemented with mostly Level 3 BLAS, we achieve perfect scaling for 2 and 4 threads and easily exceed 50% efficiency for 8 and 16 threads. This is consistent with the results presented in the related work section [8].

We exclusively use lockless data structures [40] throughout our code. This choice was dictated by fine granularity synchronization, which occurs during the pivot selection for every column of the panel and at the branching points of the recursion tree. Synchronization using mutex locks was deemed inappropriate at such frequency as it has a potential of incurring system call overhead.

Together with lockless synchronization, we use *busy waiting* on shared-memory locations to exchange information between threads using a coherency protocol of the memory subsystem. While fast in practice [8], this causes extraneous traffic on the shared-memory interconnect, which we aim to avoid. We do so by changing busy waiting for computations on independent data items. Invariably, this leads to riching the parallel granularity levels that are most likely hampered by spurious memory coherency traffic due to false sharing. Regardless of the drawback, we feel this is a satisfactory solution as we are motivated by avoiding busy waiting, which creates even greater demand for inter-core bandwidth because it has no useful work to interleave with the shared-memory polling. We refer to this optimization technique to *delayed waiting*.

Another technique we used to optimize the inter-core communication is what we call *synchronization coalescing*. The essence of this method is to conceptually group unrelated pieces of code that require a synchronization into a single aggregate that synchronizes once. The prime candidate for this optimization is the search and recording of the pivot index. Both of these operations require a synchronization point. The former needs a parallel reduction operation while the latter requires a global barrier. Neither of these are ever considered to be related to each other in the context of sequential parallelization. But with our synchronization coalescing technique, they are deemed related in the communication realm and, consequently, we implemented them in our code as a single operation.

Finally, we introduced a *synchronization avoidance* paradigm whereby we opt for multiple writes to shared memory locations instead of introducing a memory fence (and potentially a global thread barrier) to ensure global data consistency. Multiple writes are usually considered a hazard and are not guaranteed to occur in a specific order in most of the consistency models for shared memory systems. We completely side step this issue, however, as we guarantee algorithmically that each thread writes exactly the same value to memory. Clearly, this seems as an unnecessary overhead in general, but in our tightly coupled parallel implementation this is a worthy alternative to either explicit (via inter-core messaging) or implicit (via memory coherency protocol) synchronization to establish a single thread for performing the write. In short, this technique is another addition to our contention-free design.

Portability, and more precisely, performance portability, was also an important goal in our overall design. In our lock-free synchronization, we heavily rely on shared-memory consistency – a problematic feature from the portability standpoint. To address this issue reliably, we make two basic assumptions about the shared-memory hardware and the software tools. Both of which, to our best knowledge, are satisfied on majority of modern computing platforms. From the hardware perspective, we assume that memory coherency occurs at the cache line granularity. This allows us to rely on global visibility of loads and stores to nearby memory locations. What we need from the compiler tool-chain is an appropriate handling of C's volatile keyword. This, combined with the use of primitive data types that are guaranteed to be contained within a single cache line, is sufficient in preventing unintended shared-memory side effects.

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 Intel *Math Kernel Library* (MKL) are used, with layout set to row-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 lengthy description of

the DGEMM kernel, which dominates the execution time and is complex to optimize to the fullest.

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 than in the former case, and the swap operation dominates the update.

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. Although this may not be the fastest possible way of implementing the swap, when implemented like that, the swap operation 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 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 routine. This is the fastest way of implementing it, known to the authors. Because it only affects small portion of the trailing submatrix, its execution time is negligible, compared to the large DGEMM, which follows.

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 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 1024 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. As naive as this implementation is, its execution time is negligible.

3.5.4 DGEMM

The most important operation offloaded to the GPUs is the Schur complement part of the LU factorization, which is a matrix multiplication of the form $C = C - A \times B$, where A is $N \times K$, B is $K \times N$ and C is $M \times N$ and in most cases $M = N \gg K$. The value of K corresponds to the width of the panel in each step of the factorization and is commonly referred to as NB in LAPACK and PLASMA nomenclature.

The development of DGEMM presented here follows to some extent the autotuning methodology used for producing GEMM kernels for the MAGMA project [28]. There are, however, major differences: First, the code has been rewritten to operate on data in row-major layout. Second, address arithmetic has been changed to work on matrices stored in tiles. Generation and pruning of the search space has also been modified to account for tiling of input matrices, and finally, benchmarking has been done for the case of ($M = N \gg K = NB$), instead of the usual case of ($M \simeq N \simeq K$).

GPU parallelization of matrix multiplication is based on spanning the C matrix with a two-dimensional grid of *thread blocks*, where each thread block is responsible for calculating a small rectangle of the result (Figure 7). In doing so, it passes through a horizontal stripe of A and a vertical stripe of B (light gray regions on Figure7).

The dark gray rectangles of Figure 7 are magnified on Figure 8, which shows what happens inside each thread block. Here, processing follows the cycle of reading a $M_{blk} \times K_{blk}$ rectangle of A and a $K_{blk} \times N_{blk}$ rectangle of B through shared memory to registers and accumulating a $M_{blk} \times N_{blk}$ result of the multiplication in registers. At the end, when the calculation of $A \times B$ is complete, C is read, updated and written back. The light gray color on Figure 8 shows how a rectangle of C is overlaid with a two-dimensional grid of threads. The dark gray color shows the elements accessed by the first thread as it iterates through the loops.

Figure 9 shows the general structure of the kernel's pipelined loop. The loop's prologue and epilogue are marked with faded boxes, and the loop's steady state is marked with darker boxes. In this kernel, the data always passes through shared memory, what relieves the stress on the device memory and allows for efficient handling of transpositions.

First, the prologue loads the first tile of A and B to shared memory. The data is loaded to registers and deposited in shared memory. Then the code enters the steady-state loop. The loop has two stages, separated by `__syncthreads()` calls (barriers). In the first one, the data in shared memory is loaded to registers and used for calculations. At the same

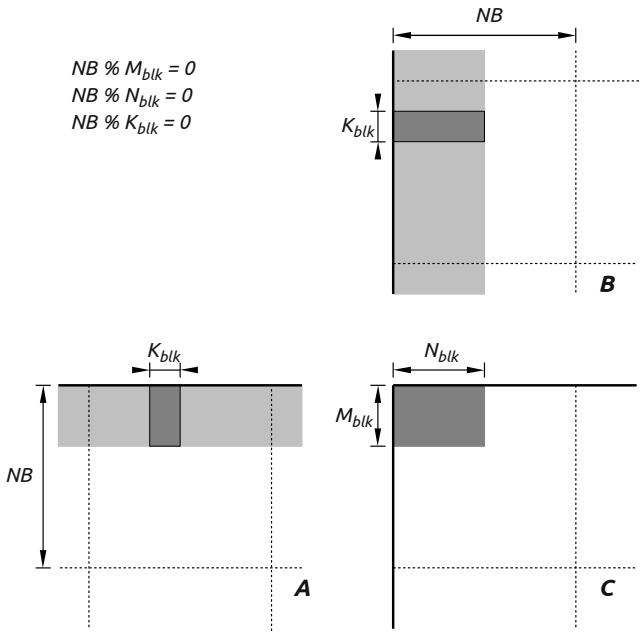

Fig. 7. Thread block's operation for the CRRB GEMM.

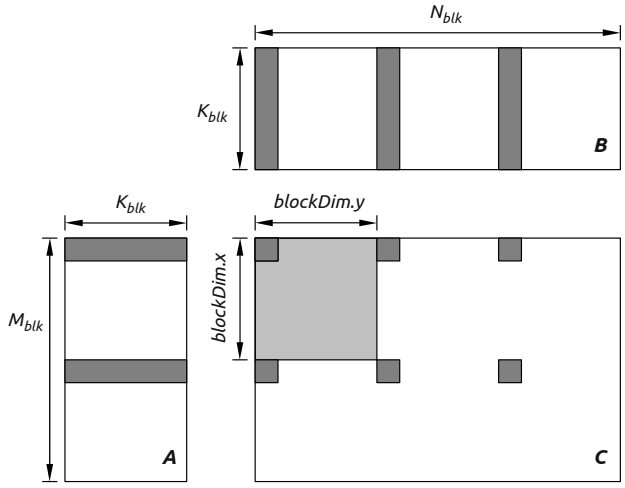

Fig. 8. CRRB GEMM at the block level.

time, new tiles of A and B are being fetched. In the second step, the newly fetched tiles are dropped to shared memory, which is than consumed as processing transitions back to step one. The code follows the classic scheme of double-buffering, where computation can be overlaid with data fetches.

The kernel is expressed as a single, heavily parametrized, source file in CUDA. Specifically, all blocking sizes are parametrized. This includes tiling for shared memory and shape of the thread block. Autotuning is used to find the best kernel parameters. It is a heuristic process based on generating a parameter search space, pruning the space using a set of constraints, and finding the fastest kernels by

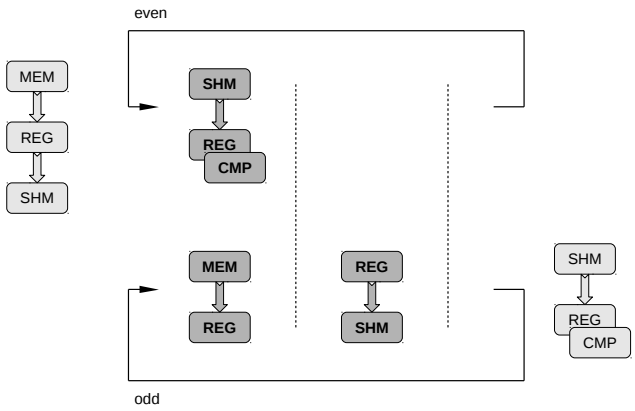

Fig. 9. The structure of the DGEMM kernel pipelined loop.

benchmarking those that pass through the selection.

The pruning engine applies straightforward heuristics to eliminate kernels, which are very unlikely to produce high performance. First of all, the kernel cannot exceed hardware capabilities, such as the number of available threads, the number of available registers, and the size of the shared memory. Kernels which do, are immediately discarded. Second, it is checked if tiling matches the shaped of the thread block, i.e., if the thread block can be shaped such that tile dimensions are divisible by thread block dimensions. It is also checked if the number of threads in a block is divisible by the warp size. Finally, three heuristic constraints are applied to further narrow the focus of the search:

- **minimum occupancy:** minimum number of threads per multiprocessor,
- **minimum register reuse:** number of FMAs per load in the innermost loop,
- **minimum number of thread blocks per multiprocessor**.

Here, minimum occupancy is set to 512 (one third of the maximum of 1,536), minimum register reuse is set to 2.0, and minimum number of thread blocks is set to 2. These choices are arbitrary and solely based on the authors' intuition. Figure 10 shows the numbers of kernels that pass through the selection for each tiling size.

Benchmarking is a crucial part of the autotuning process. The objective of pruning is to eliminate kernels which are certain to perform poorly. However, the pruning process alone has no capability to pinpointing the high performing kernels. Therefore, the performance of each kernel is measured and the fastest kernels are selected. Here, the workload of interest is the Schur complement operation in LU factorization, i.e., matrix multiplication of size $M \times N \times K$, where $M = N \gg K = NB$. All autotuning runs were done for $M = N = 12,000$ to capture the asymptotic (steady state) performance. Figure 11 shows the performance of the fastest kernels. Table 1 also shows the corresponding values of the tuning parameters. The performance is slighlty 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.

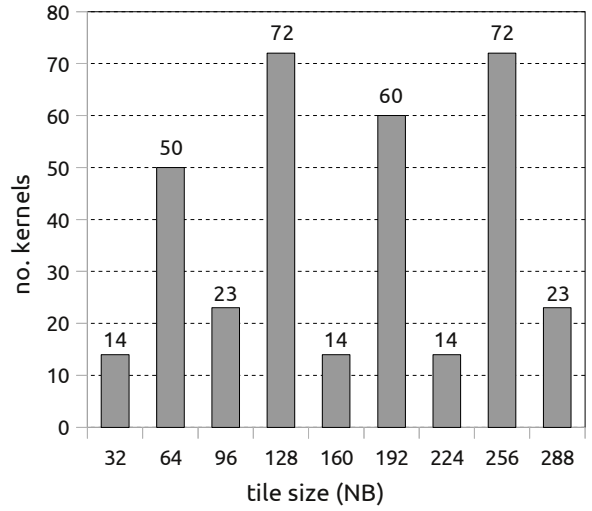

Fig. 10. Number of kernels produced for each tiling.

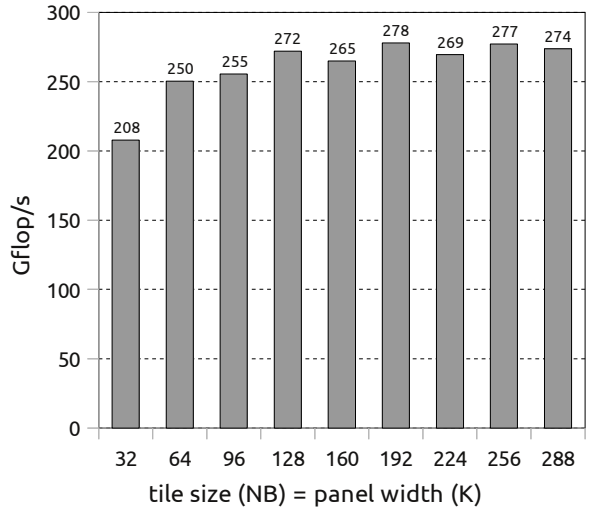

Fig. 11. Best asymptotic performance of the Schur complement operation for each tiling.

3.6 Scheduling

Manually multithreading the hybrid LU factorization would be non-trivial. 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 [46] is used for automatic dependency tracking and work scheduling. The LU factorization code is expressed with the canonical serial loop nest (Figure 12), 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 priritization. It is essential that

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
64	64×64×16	16×16	16×16	16×16	250
96	32×32×6	8×8	2×32	32×2	255
128	64×64×16	16×16	16×16	16×16	272
160	32×32×8	8×8	8×8	8×8	265
192	64×64×16	16×16	16×16	16×16	278
224	32×32×8	8×8	8×8	32×2	269
256	64×64×16	16×16	16×16	16×16	277
288	32×32×6	8×8	2×32	32×2	274

```
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. 12. Simplified QUARK code for the LU factorization.

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 queueing 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 ultra-fine 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.

3.7 Communication

Communication is shown on Figure 13. 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.

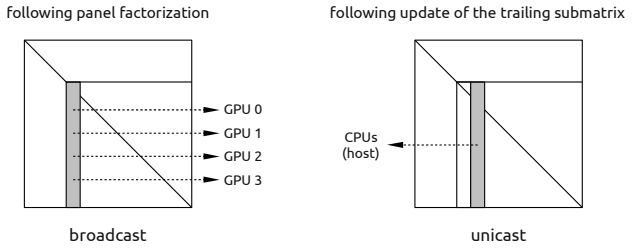

Fig. 13. CPU-GPU (host to device) communication.

4 Results

This section includes a precise description of the hardware-software environment and follows with a detailed discussion of the performance results.

4.1 Hardware & 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 4 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 $2.1\ GHz \times 12\ cores \times 4\ ops\ per\ cycle \simeq 101\ Gflop/s$, making it ∼403 Gflop/s for all four CPU sockets. The theoretical peak of a single GPU amounts to $1.147\ GHz \times 14\ cores \times 32\ ops\ per\ cycle \simeq 514\ Gflop/s$, making it ∼2055 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

Figure 14 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 manual orthogonal search, i.e., tuning tile size with all other parameters fixed, then tuning the lookahead depth, then tuning the number of cores used for the panel factorization and reiterating. The discontinuity between 23K and 25K is caused by abandoning the use of texture caches. At this point the matrix exceeds the maximum size of a 1D texture of 2^{27} (1 GB). The chart continues until 35K. Beyond that point the size of the GPU memory, with ECC protection, is exceeded (2.6 GB).

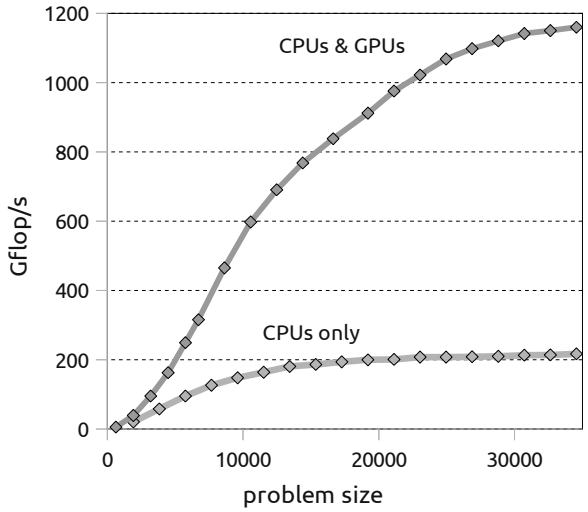

Fig. 14. Overal performance of the LU factorization.

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

Figure 15 shows a small fragment in the middle of the execution trace of the 1 Tflop/s run. In the CPU part, only the panel factorizations are shown. The entire run factors a

matrix of size 23K. The steps shown on the figure correspond to factoring submatrices of size ∼12K. 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).

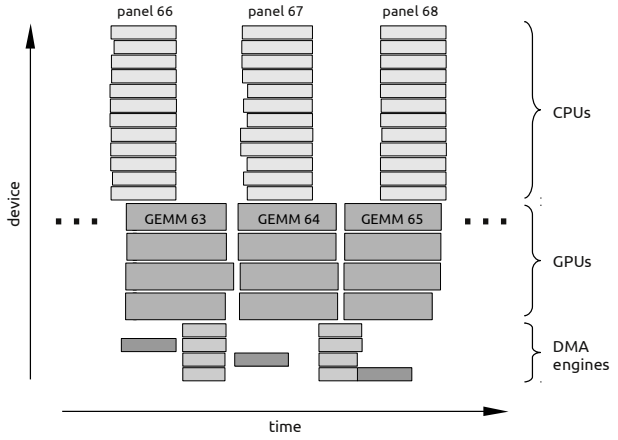

Fig. 15. A small portion in the middle of the 1 Tflop/s run.

Figure 16 shows the performance of the panel factorization throughout the entire run, using different numbers of cores, for panels of width 192. The jagged shape of the lines reflects the unpredictable nature of a cache-based CPU memory system. The black line corresponds to the 1 Tflop/s run, for which 12 cores are used (one socket). Six cores deliver inferior performance due to smaller size of their combined caches, which cannot hold tall panels at the beginning of the factorization. 24 cores deliver superior performance for tall panels, and slightly lower performance for short panels, due to increased cost of inter-socket communication. It turns out that the use of 12 cores is more efficient, even for large matrices. 12-core panel factorizations are capable of keeping up with GPU DGEMMs, while the remaining cores are committed to CPU DGEMMs. As long as panel factorizations can execute in less time than GPU DGEMMs, it is better to free up more cores to do CPU DGEMMs. At the same time, decreasing the number of panel cores to six, would quadruple the time of the initial panel factorizations (Figure 16), causing disruptions in the flow of the GPUs work (Figure 15).

Figure 17 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 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

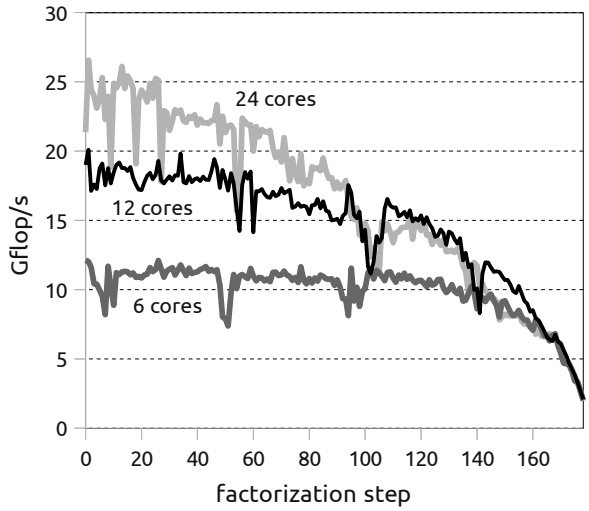

Fig. 16. Performance of panel factorization in each step of matrix factorization.

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. The two small dips on the left side of the line are due to random phenomena (jitter of unknown source).

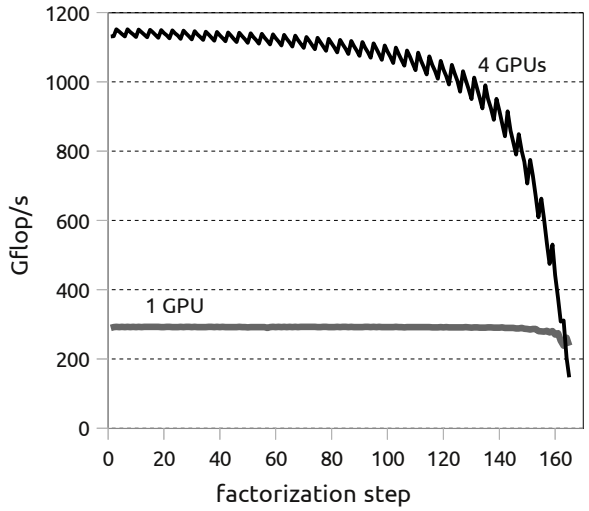

Fig. 17. Performance of DGEMM in each step of matrix factorization.

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 12 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.

Acknowledgments

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

Software

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

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009.

[2] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[3] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Computat. Pract. Exper.*, 23(2):187–198, 2011. DOI: 10.1002/cpe.1631.

[4] K. D. B. and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Applications of GPU Computing Series. Morgan Kaufmann, 2010. ISBN: 0123814723.

[5] 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 Engineering and Computer Science Department, University of Tennessee, 2008. www.netlib.org/lapack/lawnspdf/lawn200.pdf.

[6] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti. Solving dense linear systems on graphics processors. In *Proceedings of the 14th International Euro-Par Converence, EURO-PAR'08*, pages 739–748, Canary Islands, Spain, August 26-29 2008. Lecture Notes in Computer Science 5168. DOI: 10.1007/978-3-540-85451-7_79.

[7] A. M. Castaldo and R. C. Whaley. Scaling LAPACK panel operations using parallel cache assignment. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10*, Bangalore, India, January 2010. ACM. DOI: 10.1145/1693453.1693484 (submitted to ACM TOMS).

[8] A. M. Castaldo and R. C. Whaley. Scaling LAPACK panel operations using parallel cache assignment. *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 223–232, 2010.

[9] 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 Department, University of Texas at Austin, 2008. www.cs.utexas.edu/users/flame/pubs/flawn31.pdf.

[10] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine architecture and its first implementation – A performance view. *IBM J. Res. & Dev.*, 51(5):559–572, 2007. DOI: 10.1147/rd.515.0559.

[11] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic library generation for BLAS3 on GPUs. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium*, Anchorage, AK, May 16-20 2011. IEEE.

[12] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713897.

[13] 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 Engineering and Computer Science Department, University of Tennessee, 2011. http://www.netlib.org/lapack/lawnspdf/lawn259.pdf.

[14] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present and future. *Concurrency Computat.: Pract. Exper.*, 15(9):803–820, 2003. DOI: 10.1002/cpe.728.

[15] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC'05*, Seattle, WA, November 12-18 2005. IEEE Press. DOI: 10.1109/SC.2005.42.

[16] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):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. Technical Report UMINF 10.05, Department of Computer Science, Umeå University, 2010. http://www8.cs.umu.se/research/uminf/reports/2010/005/part1.pdf (submitted to ACM TOMS).

[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 Computat.: Pract. Exper.*, 2011. DOI: 10.1002/cpe.1829.

[19] W. W. Hwu, editor. *GPU Computing Gems Jade Edition*. Applications of GPU Computing Series. Morgan Kaufmann, 2011. ISBN: 0123859638.

[20] Innovative Computing Laboratory, University of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.0*, 2010. http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf.

[21] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton. Programming the Linpack benchmark for the IBM PowerXCell 8i processor. *Scientific Programming*, 17(1-2):43–57, 2009. DOI: 10.3233/SPR-2009-0278.

[22] J. Kurzak, D. A. Bader, and J. Dongarra, editors. *Scientific Computing with Multicore and Accelerators*. Chapman & Hall/CRC Computational Science Series. Taylor & Francis, 2010. ISBN: 1439825365.

[23] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equation on the CELL processor using Cholesky factorization. *Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008. DOI: TPDS.2007.70813.

[24] J. Kurzak and J. J. Dongarra. Implementation of mixed precision in solving systems of linear equations on the CELL processor. *Concurrency Computat.: Pract. Exper.*, 19(10):1371–1385, 2007. DOI: 10.1002/cpe.1164.

[25] J. Kurzak and J. J. Dongarra. QR factorization for the Cell Broadband Engine. *Scientific Programming*, 17(1-2):31–42, 2009. DOI: 10.3233/SPR-2009-0268.

[26] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency Computat.: Pract. Exper.*, 21(1):15–44, 2009. DOI: 10.1002/cpe.1467.

[27] J. Kurzak, R. Nath, P. Du, and J. J. Dongarra. An implementation of the tile QR factorization for a GPU and multiple CPUs. In *Proceedings of the State of the Art in Scientific and Parallel Computing Conference, PARA'10*, Reykjavík, June 6-9 2010. Lecture Notes in Computer Science 7133. (to appear).

[28] J. Kurzak, S. Tomov, and J. Dongarra. LAPACK working note 245: Autotuning GEMMs for Fermi. Technical Report UT-CS-11-671, Electrical Engineering and Computer Science Department, University of Tennessee, 2011. www.netlib.org/lapack/lawnspdf/lawn245.pdf (accepted to IEEE TPDS).

[29] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*, pages 884–892, Baton Roube, LA, May 25-27 2009. Lecture Notes in Computer Science 5544. DOI: 10.1007/978-3-642-01970-8_89.

[30] N. Nakasato. A fast GEMM implementation on a Cypress GPU. In *1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, PMBS'10 (held as part of SC'10)*, New Orleans, LA, November 13-19 2010. http://www.dcs.warwick.ac.uk/~sdh/pmbs10/pmbs10/Workshop_Programme_files/fastgemm.pdf.

[31] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proceedings of the 2010 International Meeting on High Performance Computing for Computational Science, VECPAR'10*, pages 83–92, Berkeley, CA, June 22-25 2010. Lecture Notes in Computer Science 6449. DOI: 10.1007/978-3-642-19328-6_10.

[32] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *Int. J. High Perf. Comput. Applic.*, 24(4):511–515, 2010. DOI: 10.1177/1094342010385729.
[33] NVIDIA. *NVIDIA CUDA C Programming Guide, Version 4.0*, 2011. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf.
[34] 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. *Comp. Graph. Forum*, 26(1):80–113, 2007. DOI: 10.1111/j.1467-8659.2007.01012.x.
[35] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM J. Res. & Dev.*, 51(5):593–604, 2007. DOI: 10.1147/rd.515.0593.
[36] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *Int. J. High Perf. Comput. Applic.*, 23(3):284–299, 2009. DOI: 10.1177/1094342009106195 .
[37] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Programming Lang. Syst.*, 20(3):483–545, 1998. DOI: 10.1145/291889.291893.
[38] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993. DOI: 10.1109/2.214440.
[39] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing, LCPC'10*, pages 136–150, Houston, TX, October 7-9 2010. Lecture Notes in Computer Science 6548. DOI: 10.1007/978-3-642-19595-2_10.
[40] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. Department of computer science, Chalmers University of Technology, Göteborg, Sweden, November 5 2004. PhD dissertation.
[41] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of the IEEE/ACM Supercomputing Conference 2011, SC'11*, Seattle, WA, November 12-18 2011. IEEE.
[42] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010. DOI: 10.1016/j.parco.2009.12.005.
[43] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proceedings of the 2010 IEEE International Parallel & Distributed Processing Symposium, IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.
[44] L. Valiant. A bridging model for parallel computation. *Communications of ACM*, 33(8):103–111, 1990.
[45] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC'08*, Austin, TX, November 15-21 2008. IEEE Press. DOI: 10.1145/1413370.1413402.
[46] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, April 2011. http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf.

Jakub Kurzak received the MSc degree in Electrical and Computer Engineering from Wroclaw University of Technology, Poland, and the PhD degree in Computer Science from the University of Houston. He is a Research Director in the Innovative Computing Laboratory in the Department of Electrical Engineering and Computer Science at the 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.

Piotr Luszczek is a Research Director of Innovative Computing Laboratory at the University of Tennessee Knoxville. His core research activity is centered around performance modeling and evaluation. He has an extensive experience through high performance numerical linear algebra and signal processing codes that achieve high efficiency on a varied array of hardware architectures such as 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 contribution to the scientific community include conference proceedings, journals, book chapters, and patent applications that show case his main research agenda expertize as well as programming paradigms, parallel language design and productivity aspects of high performance scientific computing.

Mathieu Faverge is a Post Doctoral Research Associate of Innovative Computing Laboratory at the University of Tennessee Knoxville. He received a PhD degree in Computer Science from the University of Bordeaux 1, France. 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 on hierarchical shared memory, heterogeneous and distributed systems and his contributions to the scientific community includes efficient linear algebra algorithms for those systems.

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 was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches; in 2008 he was the recipient of 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 was the recipient of the IEEE IPDPS 2011 Charles Babbage Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.