Chapter 2

Implementing Matrix Factorizations on the Cell B. E.

Jakub Kurzak
Department of Electrical Engineering and Computer Science, University of Tennessee

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

2.1 Introduction

It is clear that the impact of the multicore processors and accelerators will be ubiquitous. There are obvious advantages, however, to look at linear algebra in general and dense linear algebra in particular. This type of software is critically important to computational science across an enormous spectrum of disciplines and applications. Yet more importantly, dense linear algebra has strategic advantages as a research vehicle, because the methods and algorithms that underlie it have been so thoroughly studied and are so well understood [5, 6, 10, 17]. This chapter dissects highly optimized Cell B. E. implementations of two classic dense linear algebra computations, the Cholesky factorization and the QR factorization.

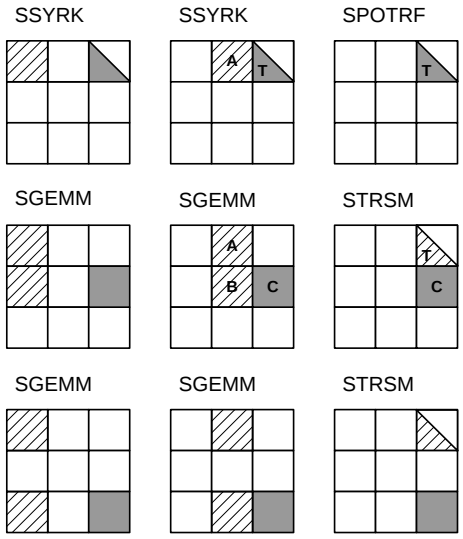

FIGURE 2.1: Tile operations in the Cholesky factorization. The sequence is left-to-right and top-down. Hatching indicates input data, shade of gray indicates in/out data.

2.2 Cholesky Factorization

The Cholesky factorization (or Cholesky decomposition) is mainly used for the numerical solution of linear equations $Ax = b$, where A is symmetric and positive definite. Such systems arise often in physics applications, where A is positive definite due to the nature of the modeled physical phenomenon. This happens frequently in numerical solutions of partial differential equations. The Cholesky factorization of an $n \times n$ real symmetric positive definite matrix A has the form

$$A = LL^T,$$

where L is an $n \times n$ real lower triangular matrix with positive diagonal elements.

The algorithm can be expressed using either the top-looking version, the left-looking version or the right-looking version. The first one follows depth-first exploration of the task graph and the last one follows the breadth-first exploration of the task graph. The left-looking variant is used here. The algorithm relies on four basic operations implemented by four computational kernels (Figure 2.1). Figure 2.2 shows the generic pseudocode of the left-looking Cholesky factorization.

```
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k] ← SSYRK(A[k][n], A[k][k])
  A[k][k] ← SPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k] ← SGEMM(A[k][n], A[m][n], A[m][k])
    A[m][k] ← STRSM(A[k][k], A[m][k])
```

FIGURE 2.2: Pseudocode of the (left-looking) Cholesky factorization.

SSYRK: The kernel applies updates to a diagonal (lower triangular) tile T of the input matrix, resulting from factorization of the tiles A to the left of it. The operation is a symmetric rank-k update.

SPOTRF: The kernel performs the Cholesky factorization of a diagonal (lower triangular) tile T of the input matrix and overrides it with the final elements of the output matrix.

SGEMM: The operation applies updates to an off-diagonal tile C of the input matrix, resulting from factorization of the tiles to the left of it. The operation is a matrix multiplication.

STRSM: The operation applies an update to an off-diagonal tile C of the input matrix, resulting from factorization of the diagonal tile above it and overrides it with the final elements of the output matrix. The operation is a triangular solve.

2.3 Tile QR Factorization

The QR factorization (or QR decomposition) offers a numerically stable way of solving underdetermined and overdetermined systems of linear equations (least squares problems) and is also the basis for the *QR algorithm* for solving the eigenvalue problem.

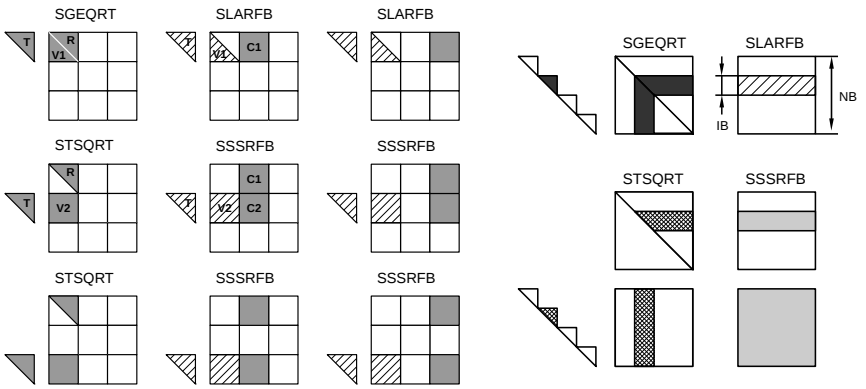

FIGURE 2.3: *Left:* Tile operations in the tile QR factorization. The sequence is left-to-right and top-down. Hatching indicates input data, shade of gray indicates in/out data. *Right:* Inner blocking in the tile QR factorization.

The QR factorization of an $m \times n$ real matrix A has the form

$$A = QR,$$

where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix. The traditional algorithm for QR factorization applies a series of elementary Householder matrices of the general form

$$H = I - \tau v v^T,$$

where v is a column reflector and τ is a scaling factor. In the block form of the algorithm a product of nb elementary Householder matrices is represented in the form

$$H_1 H_2 \ldots H_{nb} = I - V T V^T,$$

where V is an $N \times nb$ real matrix whose columns are the individual vectors v, and T is an $nb \times nb$ real upper triangular matrix [2, 16].

Here a derivative of the block algorithm is used called the *tile QR* factorization. The ideas behind the tile QR factorization are very well known. The tile QR factorization was initially developed to produce a high-performance "out-of-memory" implementation (typically referred to as "out-of-core") [11] and, more recently, to produce a high-performance implementation on "standard" (x86 and alike) multicore processors [3, 4]. The tile QR algorithm relies on four basic operations implemented by four computational kernels (Figure 2.3). Figure 2.4 shows the pseudocode of the tile QR factorization.

```
FOR k = 0..TILES-1
  A[k][k], T[k][k] ← SGEQRT(A[k][k])
  FOR m = k+1..TILES-1
    A[k][k], A[m][k], T[m][k] ← STSQRT(A[k][k], A[m][k], T[m][k])
  FOR n = k+1..TILES-1
    A[k][n] ← SLARFB(A[k][k], T[k][k], A[k][n])
    FOR m = k+1..TILES-1
      A[k][n], A[m][n] ← SSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
```

FIGURE 2.4: Pseudocode of the tile QR factorization.

SGEQRT: The kernel performs the QR factorization of a diagonal tile of the input matrix and produces an upper triangular matrix R and a unit lower triangular matrix V containing the Householder reflectors. The kernel also produces the upper triangular matrix T as defined by the compact WY technique for accumulating Householder reflectors [2, 16]. The R factor overrides the upper triangular portion of the input and the reflectors override the lower triangular portion of the input. The T matrix is stored separately.

STSQRT: The kernel performs the QR factorization of a matrix built by coupling the R factor, produced by SGEQRT or a previous call to STSQRT, with a tile below the diagonal tile. The kernel produces an updated R factor, a square matrix V containing the Householder reflectors and the matrix T resulting from accumulating the reflectors V. The new R factor overrides the old R factor. The block of reflectors overrides the square tile of the input matrix. The T matrix is stored separately.

SLARFB: The kernel applies the reflectors calculated by SGEQRT to a tile to the right of the diagonal tile, using the reflectors V along with the matrix T.

SSSRFB: The kernel applies the reflectors calculated by STSQRT to two tiles to the right of the tiles factorized by STSQRT, using the reflectors V and the matrix T produced by STSQRT.

A naive implementation, where the full T matrix is built, results in 25 % more floating point operations than the standard algorithm. In order to minimize this overhead, the idea of *inner-blocking* is used, where the T matrix has sparse (block-diagonal) structure (Figure 2.3) [7–9].

2.4 SIMD Vectorization

The keys to maximum utilization of the synergistic processing elements (SPEs) are highly optimized implementations of the computational kernels, which rely on efficient use of the short-vector single instruction multiple data (SIMD) architecture. For the most part, the kernels are developed by applying standard loop optimization techniques, including tiling, unrolling, reordering, fusion, fission, and sometimes also collapsing of loop nests into one loop spanning the same iteration space with appropriate pointer arithmetics. Tiling and unrolling are mostly dictated by Local Store latency and the size of the register file, and aim at hiding memory references and reordering of vector elements, while balancing the load of the two execution pipelines. Due to the huge size of the SPEs' register file, unrolling is usually quite aggressive.

Implementation of the tile kernels assumes a fixed size of the tiles. Smaller tiles (finer granularity) have a positive effect on scheduling for parallel execution and facilitate better load balance and higher parallel efficiency. Bigger tiles provide better performance in sequential execution on a single SPE. In the case of the CELL chip, the crossover point is rather simple to find for problems in dense linear algebra. From the standpoint of this work, the most important operation is matrix multiplication in single precision. It turns out that this operation can achieve the peak performance of the SPE for matrices of size 64×64 (see the preceeding chapter). The fact that the peak performance can be achieved for a tile of such a small size has to be attributed to the large size of the register file and fast access to the Local Store, undisturbed with any intermediate levels of memory. Also, such a matrix occupies a 16 KB block of memory, which is the maximum size of a single DMA transfer. Eight such matrices fit in half of the Local Store providing enough flexibility for multibuffering while, at the same time, leaving enough room for the code.

Table 2.1 shows characteristics of the Cholesky kernels and the tile QR kernels. It can be observed that the Cholesky kernels required moderate effort. Initially, all the kernels were coded using C language SIMD extensions (intrinsics) and required roughly 300 lines of code per kernel. However, preprocessor macros were used and the resulting assembly code is significantly longer. Nevertheless, the effort associated with development and maintenance of this code is rather small. At the same time, the delivered performance is more than satisfactory. Specifically, the SGEMM and SSYRK kernels deliver 90 and 79% of the peak respectively, which has to be considered quite good for SIMD code developed in a higher level language. The STRSM kernel delivers poorer performance due to a lower level of SIMD parallelism available and the SPOTRF kernel performs the poorest for the same reason. The SPOTRF kernel simply performs the Cholesky factorization within a tile and is the the most complex operation to SIMD'ize with the lowest level of available SIMD parallelism.

TABLE 2.1: Complexity and performance characteristics of SPE micro-kernels for the Cholesky factorization (top) and the tile QR factorization (bottom). Bold font highlights the largest codes and the highest performance. All operations are for matrices of size 64×64 (n=64).

Kernel Name	Lines of Code in C	Lines of Code in ASM	Object Size [KB]	Exec. Time [μs]	Flop Count Formula	Exec. Rate [Gflop/s]	Fraction of Peak [%]
SGEMM$_C$	330	2000	7.8	23	$2n^3$	23.03	90
SGEMM$_{ASM}$	-	**3900**	6.2	22	$2n^3$	**24.04**	94
SSYRK	160	1000	3.6	13	n^3	20.11	79
STRSM	310	1600	6.2	16	n^3	16.26	64
SPOTRF	340	800	3.1	14	$^1/_3n^3$	6.57	26
SSSRFB	1600	2200	8.8	47	$4n^3$	**22.20**	87
STSQRT	1900	**3600**	14.2	46	$2n^3$	11.40	45
SLARFB	600	600	2.2	41	$2n^3$	12.70	50
SGEQRT	1600	2400	9.0	57	$^4/_3n^3$	6.15	24

Table 2.1 also includes the SGEMM kernel developed in the SPE assembly language, which was described in the previous chapter. In this case the effort was rather huge and involved development of 3900 lines of hand-tuned assembly code. At the same time, the performance gain is less than 5 %. Such an effort is justified in reasearch circles, but would be questionable in commercial environments. Nevertheless, the performance for parallel runs, presented further in the text, relies on the fast assembly kernel.

It should be pointed out that the performance of the kernels developed in the C language is very sensitive to the version of the compiler used and the compilation flags. The authors exhaustively tried all the combinations and the table reports the best results achieved. Many times high performance was only achievable while using the spu-gcc, version 3.4.1, released in SDK 1.1, toolchain 2.3. Most of the time either the flag -O3 or the flag -Os delivered the best performance. Since the follow up versions of the compiler delivered poorer performance for the kernels, the code posted online by the authors (2.9) includes the kernels precompiled to assembly using the old compiler.

The development of the Cholesky kernels was moderately difficult. Three of them implement simple Level 3 BLAS operations, while the fourth one implements the Cholesky factorization on a tile, which is not overly complicated. The same cannot be claimed about the tile QR factorization kernels. None of the kernel operations is a simple BLAS operation, and the technique of inner-blocking further complicates matters.

Inner-blocking in the tile QR algorithm is required to minimize the number of extraneous floating-point operations (beond the $^4/_3n^3$ formula) coming from the accumulation of Householder reflectors. As necessary as the

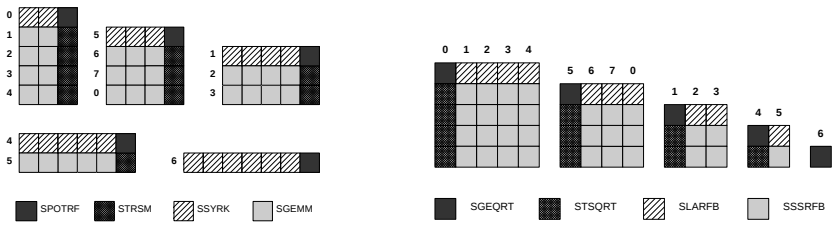

FIGURE 2.5: Assignment of work to SPEs for the Cholesky factorization (left) and the tile QR factorization (right).

inner-blocking is, it also restricts the level of available SIMD parallelism. Ideally, the size of the inner block would be chosen in the process of autotuning. However, such an approach would require some means of automatic code generation. Since such capabilities were not available here, the size was picked arbitrarily. For productivity reasons, the size of four elements was picked to match the size of the SIMD vector length in single precision.

As Table 2.1 shows, it was also possible to achieve good performance for the tile QR kernels coded in the C language using intrinsics. Most importantly, good performance was achieved for the SSSRFB kernel, which is as performance-critical to the tile QR factorization as the SGEMM kernel performance is critical to the Cholesky factorization. At the same time, much heavier coding effort was involved, resulting in three kernels larger that 1500 lines of source code (SGEQRT, SSSRFB, STSQRT) and the STSQRT kernel ultimately translating to 3600 lines of assembly code.

2.5 Parallelization—Single Cell B. E.

Matrix factorizations represent computations with a very clear structure and regular data access pattern. This motivates the use of static partitioning of work to the SPEs, shown in Figure 2.5. For the Cholesky factorization, in each step of the factorization, each SPE goes through one row of tiles. The assignment of rows is cyclic, from step to step, and the SPE which "runs out of work" in a given step immediately follows to the consecutive step, a behavior resembling the popular technique of *lookahead*. The scheme is followed for the tile QR factorization, except here each SPE goes through one column of tiles.

Due to the regular nature of these workloads, static scheduling is extemely straighforward to implement. Using a simple formula on tiles' idexes, each SPE can traverse its own path through the iteration space. Additionally, at each step, a check for data dependencies is required. The SPE does that by looking

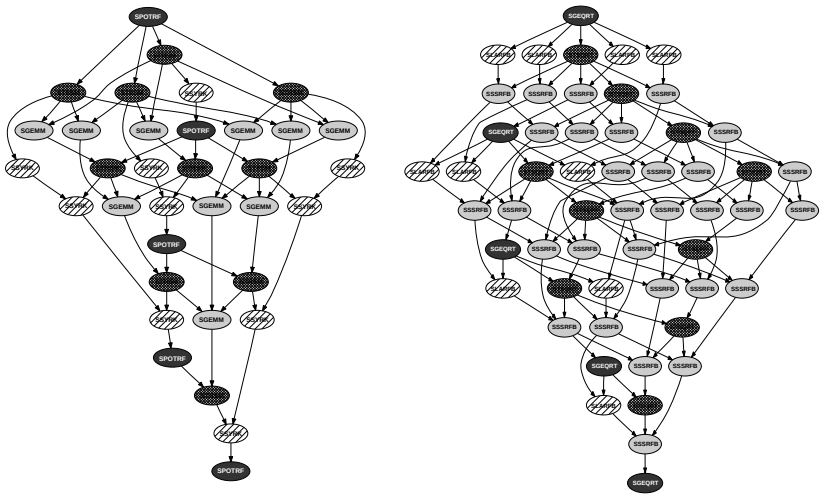

FIGURE 2.6: Direct Acyclic Graphs of the Cholesky factorization (left) and the tile QR factorization (right) for a matrix of size 5×5 tiles.

up a progress table in its Local Store. The progress table contains the global progress information and is replicated on all SPEs. The progress table holds one entry (a byte) for each tile of the input matrix, indicating progress of the computation associated with that tile. At the completion of each operation, the SPE broadcasts the progress information to progress tables of all SPEs with an SPE-to-SPE DMA.

Alternatively to the static scheduling, a dynamic scheduling could be used, based on representing the computation as a task graph or *Direct Acyclic Graph* (DAG). The task is rather non-trivial due to the complexity of the DAGs of dense matrix factorizations (Figure 2.6). One framework capable of such scheduling on the Cell B. E. is the *Cell Superscalar* (CellSs) project from the Barcelona Supercomputer Center [1, 15]. Unfortunately, due to the overheads of dynamically scheduling complex DAGs, the software is still not competitive, in terms of performance, with the approach presented here.

An important aspect of the algorithm is overlapping of communication and computation by double-buffering of data. At each step, the tiles of the input matrix are exchanged between the main memory and Local Store. Since scheduling is static, upcoming operations can be anticipated and the necessary data prefetched. In fact, all data buffers are duplicated and, at each operation, a prefetch of data is initiated for the upcoming operation (again, subject to a dependency check). If the prefetch fails for dependency reasons, data are fetched in a blocking mode right before the operation. Algorithm 1 shows the mechanism of double-buffering in matrix factorizations.

The pipelined scheduling scheme along with double-buffering of data trans-

Algorithm 1 General scheme of double-buffering in the Cholesky and tile QR factorizations.

```
1: while more work to do do
2:    if data not prefetched then
3:       wait for dependencies
4:       fetch data
5:    end if
6:    if more work to follow then
7:       if dependencies met then
8:          prefetch data
9:       end if
10:   end if
11:   compute
12:   swap buffers
13: end while
```

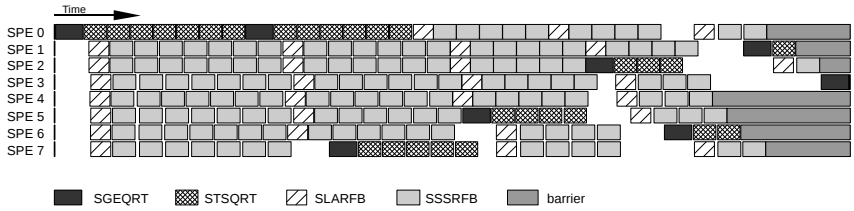

FIGURE 2.7: Execution trace of the tile QR factorization of a 512×512 matrix. (total time: 1645 μs, execution rate: 109 Gflop/s).

fers provide for smooth execution with minimal idle time caused by dependency stalls and almost no time lost to data transfers. This is clearly visible on a trace of the tile QR factorization presented in Figure 2.7.

2.6 Parallelization—Dual Cell B. E.

Given the single-Cell B. E. implementation, extension to dual-Cell B. E. implementation (e.g., IBM QS20, IBM QS22) is relatively straightforward. A single PPE process can launch 16 SPE threads, eight on each Cell B. E. The single-Cell B. E. code is going to run correctly on a dual-Cell B. E. system by simply increasing the number of SPEs to 16.

The problem is a one of performance of the memory system. The dual-Cell blades are *Non-Uniform Memory Access* (NUMA) architectures. Each Cell

B. E. is associated with a separate memory node. Peak bandwidth to the local node is 25.6 GB/s. Cross-traffic, however, is handled at a much lower bandwidth (roughly half of that number). It is important, then, that each SPE satisfies its data needs mostly from the local memory node.

This situation is addressed by duplicating the input matrix in both memory nodes (*libnuma* is used for correct memory placement). Each SPE reads data only from the local node, but writes data to both nodes. From the perspective of the shared memory model, it can be viewed as a manual implementation of the write-back memory consistency protocol. From the perspective of a distributed memory model, it can be viewed as non-blocking collective communication (broadcast) or as one-sided communication. The obvious limitation is that the approach would not be scalable to larger NUMA systems. As of today, however, larger Cell-based NUMA systems do not exist.

One technical detail to be mentioned here is the acknowledgment DMAs implementing the synchronization protocol between SPEs. When 16 SPEs are used, each SPE needs to send 16 acknowledgment messages following a write of data to the system memory. The acknowledgment DMA is fenced with the data DMA and the SPE also sends such a message to its own progress table (hence 16 messages are sent and not 15). The DMA request queue is, however, only 16 entries deep, and issuing 16 acknowledgment requests at the same time stalls data transfers until some requests clear the queue. A simple remedy is the use of a DMA list with 16 elements, where the elements point to appropriate Local Store locations of the other SPEs. The code alternates between two such lists in the double-buffered communication cycle.

2.7 Results

Results presented here are produced by the two 3.2 GHz Cell B. E. chips of the QS20 dual-socket blade running Fedora Core 7 Linux. The code is cross-compiled using x86 SDK 3.1, although the kernels are cross-compiled with an old x86 SPU GCC 3.4.1 cross-compiler, since this compiler yields the highest performance. It also needs to be mentioned that the implementation utilizes *Block Data Layout* (BDL) [13, 14], where each tile is stored in a continuous 16 KB portion of the main memory, which can be transferred in a single DMA, which puts an equal load on all 16 memory banks. Tiles are stored in the row-major order, and also data within tiles are arranged in the row-major order, a common practice on the Cell B.E. Translation from standard (FORTRAN) layout to BDL can be implemented very efficiently on the Cell B.E. [12]. Here the translation is not included in timing results. Also, in order to avoid the problem of TLB misses, all the memory is allocated in huge TLB pages and"faulted in" at initialization. As a result, an SPE never incurs

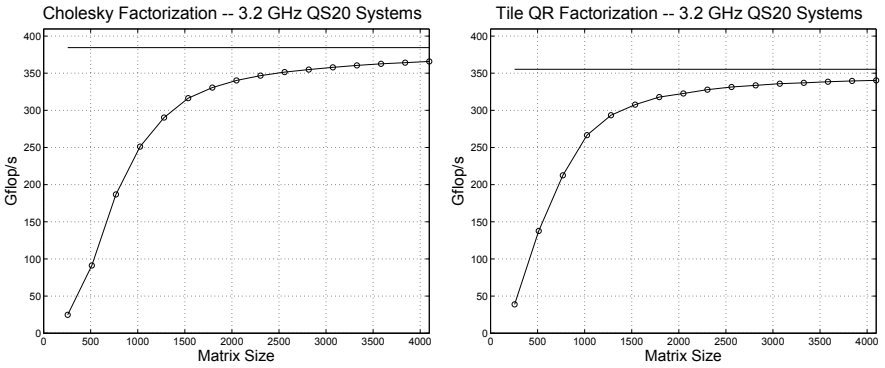

FIGURE 2.8: Performance of the Cholesky factorization (left) and the tile QR factorization (right) in single precision on an IBM QS20 blade using two Cell B. E. chips (16 SPEs). Square matrices were used. The solid horizontal line marks the performance of the SGEMM kernel for the Cholesky factorization, and the SSSRFB kernel for the tile QR factorization, multiplied by the number of SPEs (16).

a TLB miss during the run. Correct NUMA memory placement is enforced using the *libnuma* library.

Figure 2.8 and Tables 2.2 and 2.3 show the performance. Not only do the factorizations get close to the peak performance of the hardware, but also the performance curves raise very quickly with the sizes of the matrices, i.e., the code delivers very good performance even for relatively small problem sizes. Ultimately the algorithm's performance is limited by the performance of the critical SPE kernels, SGEMM for Cholesky and SSSRFB for tile QR.

2.8 Summary

It has been shown here that a silicon chip can provide an outstanding performance for compute-intensive scientific workloads by combining short-vector SIMD capabilities with multicore architecture and also providing for explicit control over caches (Local Stores). It is also an important factor that the SPEs allow for implementation of complex synchronization mechanisms and thus for efficiently exploiting task-level parallelism in workloads with complex data dependencies, such as dense matrix factorizations. One point to be made here is that successful implementation relies on addressing all aspects of performance optimization: exploiting data-level parallelism through short-vector SIMD vectorization, exploiting task-level parallelism through SPE-parallelization and

TABLE 2.2: Performance of the Cholesky factorization in single precision on two 3.2 GHz Cell B. E. chips of the IBM QS20 dual-socket blade (16 SPEs).

Matrix Size	Execution Rate [Gflop/s]	Fraction of Peak [%]	Fraction of SGEMM Peak [%]
256	24	6	6
512	91	22	24
768	186	46	49
1024	251	61	65
1280	290	71	75
1536	316	77	82
1792	330	81	86
2048	340	83	88
3072	357	87	93
4096	365	89	95

TABLE 2.3: Performance of the tile QR factorization in single precision on two 3.2 GHz Cell B. E. chips of the IBM QS20 dual-socket blade (16 SPEs).

Matrix Size	Execution Rate [Gflop/s]	Fraction of Peak [%]	Fraction of SSSRFB Peak [%]
256	38	9	11
512	137	34	39
768	212	52	60
1024	266	65	75
1280	293	72	83
1536	307	75	87
1792	317	78	90
2048	322	79	91
3072	335	82	95
4096	340	83	96

exploiting the memory hierarchy through explicit control of the local memories.

2.9 Code

The code is freely available under the BSD license and can be downloaded from the author's web site `http://icl.cs.utk.edu/~kurzak/`. Although the authors put a lot of effort into making the code both robust and readable, it is a proof-of-concept prototype and not a production-quality code.

Bibliography

[1] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Tampa, FL, November 11-17 2006. ACM. DOI: 10.1145/1188455.1188546.

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

[3] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008. DOI: 10.1002/cpe.1301.

[4] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput. Syst. Appl.*, 35:38–53, 2009. DOI: 10.1016/j.parco.2008.10.002.

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

[6] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998. ISBN: 0898714281.

[7] E. Elmroth and F. G. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In *Applied Parallel Computing, Large Scale Scientific and Industrial Problems, 4th International Workshop, PARA'98*, Umeå, Sweden, June 14-17 1998. *Lecture Notes in Computer Science* 1541:120-128. DOI: 10.1007/BFb0095328.

[8] E. Elmroth and F. G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. & Dev.*, 44(4):605–624, 2000.

[9] E. Elmroth and F. G. Gustavson. High-performance library software for QR factorization. In *Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, 5th International Workshop, PARA 2000*, Bergen, Norway, June 18-20 2000. *Lecture Notes in Computer Science* 1947:53–63. DOI: 10.1007/3-540-70734-4_9.

[10] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996. ISBN: 0801854148.

[11] B. C. Gunter and R. A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, 2005. DOI: 10.1145/1055531.1055534.

[12] 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.

[13] N. Park, B. Hong, and V. K. Prasanna. Analysis of memory hierarchy performance of block data layout. In *Proceedings of the 2002 International Conference on Parallel Processing, ICPP'02*, pages 35–44, Vancouver, Canada, August 18-21 2002. IEEE Computer Society. DOI: 10.1109/ICPP.2002.1040857.

[14] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):640–654, 2003. DOI: 10.1109/TPDS.2003.1214317.

[15] 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.

[16] R. Schreiber and C. van Loan. A storage-efficient WY representation for products of Householder transformations. *J. Sci. Stat. Comput.*, 10:53–57, 1991.

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