# QR Factorization for the CELL Processor

– LAPACK Working Note 201

**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, University of Manchester

## ABSTRACT

**The QR factorization is one of the most important operations in dense linear algebra, offering a numerically stable method for solving linear systems of equations including overdetermined and underdetermined systems. Classic implementation of the QR factorization suffers from performance limitations due to the use of matrix-vector type operations in the phase of panel factorization. These limitations can be remedied by using the idea of updating of QR factorization, rendering an algorithm, which is much more scalable and much more suitable for implementation on a multi-core processor. It is demonstrated how the potential of the CELL processor can be utilized to the fullest by employing the new algorithmic approach and successfully exploiting the capabilities of the CELL processor in terms of Instruction Level Parallelism and Thread-Level Parallelism.**

**KEYWORDS:** CELL processor, multi-core, numerical algorithms, linear algebra, matrix factorization

## 1 Introduction

State of the art, numerical linear algebra software utilizes *block algorithms* in order to exploit the memory hierarchy of traditional cache-based systems [1, 2]. Public domain libraries such as LAPACK [3] and ScaLAPACK [4] are good examples. These implementations work on square or rectangular submatrices in their inner loops, where operations are encapsulated in calls to *Basic Linear Algebra Subroutines* (BLAS) [5], with emphasis on expressing the computation as level 3 BLAS (*matrix-matrix* type) operations.

The fork-and-join parallelization model of these libraries has been identified as the main obstacle for achieving scalable performance on new processor architectures. The arrival of multi-core chips increased the demand for new algorithms, exposing much more thread-level parallelism of much finer granularity. This paper presents an implementation of the QR factorization based on the idea of updating the QR factorization. The algorithm, referred to as *tile QR*, processes the input matrix by small square blocks of fixed size, providing for great data locality and fine granularity of parallelization. In the case of the CELL processor, it also readily solves the problem of limited size of private memory associated with each computational core.

Section 2 provides a brief discussion of related work. Section 3 presents a short description of the algorithm. Section 4 gives a quick overview of processor architecture, followed by a discussion of vectorization and parallelization of the code. Sections 5, 6 and 7 follow with the presentation of the performance results, conclusions and possibilities for future developments.

This article focuses exclusively on the aspects of efficient implementation of the algorithm and makes

no attempts at discussing the issues of numerical stability of the algorithm, nor issues stemming from the use of single precision with truncation rounding, and lack of support for NaNs and denorms (which is the way the CELL processor implements single precision floating point operations).

## 2 Related Work

The first experiences with implementing dense matrix operations on the CELL processor were reported by Chen et al. [6]. Performance results were presented for single precision matrix multiplication and the solution of dense systems of linear equations in single precision using LU factorization. The authors of this article refined this work by using the LU factorization in single precision along with the technique of iterative refinement to achieve double precision accuracy of the final solution [7].

Cholesky factorization was identified as an algorithm rendering itself easily to formulation as *algorithm by tiles*. It was subsequently implemented, delivering parallel scaling far superior to that of LU (in its classic form). A mixed-precision iterative refinement technique was used to solve symmetric positive definite systems of equations, producing results with double precision accuracy while exploiting the speed of single precision operations [8].

Other developments worth noting were further refinements of the work on optimizing the matrix multiplication, first by Hackenberg [9, 10] and then by Alvaro et al. [11]. It is also worthwhile to note that impressive performance was achieved by Williams et al. for sparse matrix operations on the CELL processor [12].

A great body of work has been devoted to using orthogonal transformations for matrix factorization. Therefore, it would be hard to give a comprehensive overview. Important work on parallelizing the QR factorization on distributed memory multiprocessors was done by Pothem and Raghavan [13] and Chu and George [14]. Berry et al. successfully applied the idea of using orthogonal transformations to annihilate matrix elements by tiles, in order to achieve a highly parallel distributed memory implementation of matrix reduction to the block upper-Hessenberg form [15].

One of the early references discussing methods for updating matrix factorizations is the paper by Gill et al. [16]. Gunter and van de Geijn employed the idea of updating the QR factorization to implement an efficient out-of-core (out-of-memory) factorization [17] and also introduced the idea of inner blocking for reducing the amount of extra floating point operations. Buttari et al. [18] identified the potential for the out-of-memory approach for "standard" (x86 and alike) multi-core processors and reported results for QR, LU and also Cholesky factorizations.

This article is a result of the convergence of work on implementing dense matrix operations on the CELL processor and the work on new, scalable dense linear algebra algorithms for multi-core architectures.

## 3 Algorithm

The tile QR algorithm is very well documented in the literature [17, 18]. The algorithm produces the same $R$ factor as the classic algorithm (e.g., the implementation in the LAPACK library), but a different set of Householder reflectors, which requires a different procedure to build the $Q$ matrix. Whether the $Q$ matrix is actually needed depends on the application.

The algorithm relies on four basic operations implemented by four computational kernels (Figure 1). Here the LAPACK-style naming convention, introduced by Buttari et al. [18], is followed. The capital letter $S$ at the beginning indicates the use of single precision.

**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 [19, 20]. 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.

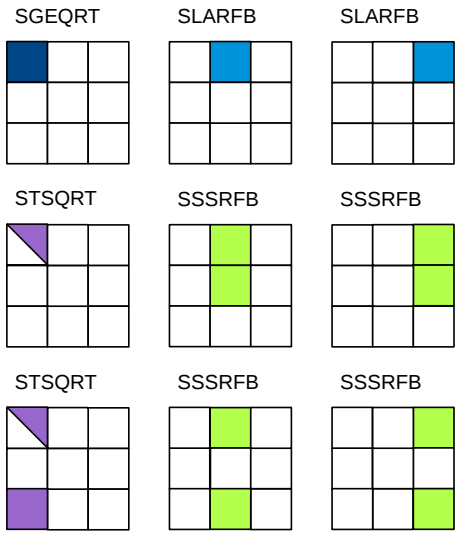


Figure 1: Basic operations of the tile QR factorization.

**STSQRT:** The kernel performs the QR factorization of a matrix built by coupling an $R$ 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.

LAPACK-style block QR factorization relies on the compact $WY$ technique for accumulating Householder reflectors in order to express computation in terms of level 3 BLAS (matrix-vector) operations. The technique requires calculation of a square matrix $T$ per each panel of the input matrix, where the size of $T$ is equal to the width of the panel and, most of the time, much smaller than the height of the panel. In this case, the overhead associated with manipulating the $T$ matrices is negligible.

In a naive implementation of the tile QR factorization, a $T$ matrix is produced for each square tile of the panel and used in updating tiles to the right of each tile of the panel. This approach results in 25 % more operations than the standard QR algorithm.

It can be observed, however, that in principle the updating algorithm can be implemented relying on level 2 BLAS (matrix-vector) operations, without the use of the $T$ matrices and associated overheads. Interestingly, in such case, the updating algorithm results in the same number of floating point operations as the standard QR algorithm ($2MN^2 - {}^2/_3 N^3$). Obviously, such implementation has to perform poorly due to the memory-bound nature of level 2 BLAS.

The key to achieving performance is to find the right trade-off between extra operations and memory intensity. This can be achieved by implementing the tile operations using the block algorithms within the tile. With internal block size much smaller than tile size, resulting $T$ matrices are not "full" upper triangular matrices, but instead consist of upper triangular blocks along the diagonal of size equal to the inner block size (Figure2).

# 4 Implementation

The process of implementing the algorithm on the CELL processor included a few design choices (some of them arbitrary), which the authors would like to discuss here.

The tile size of 64×64 is a common practice for implementing dense matrix operations in single precision on the CELL processor. It has been shown that at this size matrix multiplication kernels can achieve over 99 % of the SPE peak [11]. At the same time, the DMA transfer of a single tile fully utilizes the memory system consisting of 16 banks interleaved on a cache line boundary of 128 bytes.

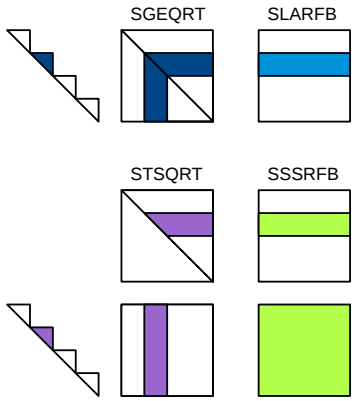


Figure 2: Inner blocking of the tile operations.

Typically, the inner block size is chosen using some method of auto-tuning. In this case the inner block size of 4 has been chosen arbitrarily, mostly for coding simplicity stemming from the size of the SIMD vector of four single precision floating point elements. It turs out, however, that even at such a small size, the code does not become memory-bound thanks to the small, flat latency of the Local Store. It also introduces an acceptable amount of extra floating point operations. It is very unlikely that a different choice would yield significantly better results.

Finally, it has been chosen to implement the SGEQRT and STSQRT kernels using LAPACK-style block algorithm internally within the kernels. Potentially, the tile algorithm could also be used inside the kernels. Such approach would, however, dramatically complicate the application of the reflectors. The update operation could not be implemented efficiently.

## 4.1 CELL Architecture Overview

The CELL processor has been available since 2005 and is well known to the numerical computing community. It is not, however, a main-stream solution and is often perceived as a special-purpose accelerator device. As a result, the authors restrain from an extensive overview of the architecture, but do introduce the basic CELL vocabulary, and the highlights of the chip computing core design.

The CELL processor is an innovative multi-core architecture consisting of a standard processor, the *Power Processing Element* (PPE), and eight short-vector, *Single Instruction Multiple Data* (SIMD) processors, referred to as the *Synergistic Processing Elements* (SPEs). The SPEs are equipped with *scratchpad memory* referred to as the *Local Store* (LS) and a *Memory Flow Controller* (MFC), to perform *Direct Memory Access* (DMA) transfers of code and data between the system memory and the Local Store.

The core of the SPE is the *Synergistic Processing Unit* (SPU). The SPU is a RISC-style SIMD processor featuring 128 general purpose registers and 32-bit fixed-length instruction encoding. An SPU implements instructions to perform single and double precision floating point arithmetics, integer arithmetics, logicals, loads and stores, compares and branches. SPU's nine execution units are organized into two pipelines, referred to as the odd and even pipeline. Instructions are issued in-order, and two independent instructions can be issued simultaneously if they belong to different pipelines (what is referred to as *dual-issue*).

SPU executes code form the Local Store and operates on data residing in the Local Store, which is a fully pipelined, single-ported, 256 KB of *Static Random Access Memory* (SRAM). Load and store instructions are performed within local address space, which is untranslated, unguarded and noncoherent with respect to the system address space. Loads and stores transfer 16 bytes of data between the register file and the Local Store and complete with fixed six-cycle delay and without exception.

## 4.2 SIMD Vectorization

The keys to maximum utilization of the SPEs are highly optimized implementations of the computational kernels, which rely on efficient use of the short-vector 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 SPU's register file, unrolling is usually quite extensive.

Most of the techniques used to build the tile QR kernels are similar to those used to build the Cholesky factorization kernels [8] and the high performance SGEMM (matrix multiplication) kernels [11]. The main difference here is the use of inner blocking, which substantially narrows down the design choices. Most importantly, the use of inner blocking imposes the structure of nested loops, where a single iteration of the outermost loop implements a single block operation. For instance, one iteration of the outermost loop of SGEQRT and STSQRT produces a block of four reflectors and the associated 4×4 upper triangular block of $T$; one iteration of the outermost loop of SLARFB and SSSRFB applies a block of four reflectors and utilizes a 4×4 block of $T$.

All kernels are written in C using mostly SIMD language extensions (intrinsics) and sometimes inline assembly. Table 1 shows the size of the C source code, assembly code and object code of the kernels. Since all the code is hand-written, it gives some idea of its complexity. Table 2 reports the performance of the kernels in terms of Gflop/s and percentage of the peak (of a single SPE). The authors are only able to achieve this performance while compiling the kernels with SPU GCC 3.4.1. The paragraphs that follow briefly discuss technicalities related to each of the kernels.

The SSSRFB kernel, being the most performance-critical (contributing the most floating point operations), is optimized the most. This kernel actually allows for the most extensive optimizations, since all loops have fixed boundaries. Therefore, the technique of collapsing loop nests into one loop is used here, along with double-buffering, where odd and even iterations overlap each other's arithmetic operations with loads, stores and vector element permutations. Also, input arrays are constrained with 16 KB alignment, and pointer arithmetic is implemented by calculating data offsets from the iteration variable (loop counter) by using bit manipulation. It turns out that all these operations can be implemented using quadword shifts, rotations and shuffles, and placed in the odd pipeline, where they can be hidden behind floating point arithmetics. Interestingly, it also turns out that for some loops, mostly rearranging vector elements, shuffles can be replaced with bit select operations to yield more balanced odd and even pipeline utilization.

Table 1: Complexity characteristics of tile QR SPE micro-kernels. (Bold font indicates the most complex kernel.)

| Kernel Name | Lines of Code in C[a] | Lines of Code in ASM[b] | Object Size [KB][c] |
|---|---|---|---|
| SSSRFB | 1,600 | 2,200 | 8.8 |
| **STSQRT** | **1,900** | **3,600** | **14.2** |
| SLARFB | 600 | 600 | 2.2 |
| SGEQRT | 1,600 | 2,400 | 9.0 |
| **Total** | 5,700 | 8,800 | 34.2 |

[a] size of code in C before or after preprocessing, whichever is smaller

[b] size of code in assembly after removing the .align statements

[c] sum of .text and .rodata sections (not size of the .o file)

Table 2: Performance charecteristics of tile QR SPE micro-kernels. (Bold font indicates the most performance-critical kernel.)

| Kernel Name | Exec. Time [$\mu$s][a] | Flop Count Formula[b] | Exec. Rate [Gflop/s][c] | Fraction of Peak [%][a,d] |
|---|---|---|---|---|
| **SSSRFB** | **47** | $\mathbf{4b^3}$ | **22.16** | **87** |
| STSQRT | 46 | $2b^3$ | 11.40 | 45 |
| SLARFB | 41 | $2b^3$ | 12.70 | 50 |
| SGEQRT | 57 | $^4/_3 b^3$ | 6.15 | 24 |

[a] values are rounded

[b] tile size b = 64

[c] values are truncated

[d] single SPE

In principle, the SSSRFB kernel shares many properties with the SGEMM kernel, and one could expect performance similarly close to the peak (99.8 % was reported for SGEMM [11]). This is not the case for a few reasons. The main contributor of performance loss of the SSSRFB kernel is the prologue and epilogue code of the inner loops, which cannot be hidden behind useful work. Also, the reported performance of the SSSRFB kernel cannot reach the peak because of the extra operations, related to the application of the $T$ matrix, which are not accounted for in the standard formula for operation count, $4b^3$. The actual number of operations is $4b^3 + sb^2$, where s is the size of internal blocking.

The STSQRT kernel has been identified as the second most critical for performance. STSQRT produces data, which is consumed by many SSSRFB kernels in parallel. As a result, it is important, in the context of parallel scheduling, that the STSQRT kernel executes in a shorter time than the SSSRFB kernel. This task proved quite difficult and the STSQRT kernel took significant coding effort and results in the longest code. One fact that is taken advantage of is that, at each step (each outer loop iteration), a block of reflectors of the same size is produced (64×4). This allowes for performing the panel factorization (production of four reflectors) to be executed entirely in the register file, using 64 registers. First, the panel is loaded, then four steps are performed, each producing one reflector and applying it to the rest of the panel, then the panel is stored. The whole procedure is completely unrolled to one block of *straight-line* code.

As extreme as it might seem, this step alone proves to be insufficient to deliver the desired performance. The operations applying the panel to the remaining submatrix have to also be extensively optimized by heavy unrolling and addressing of special cases (e.g., different treatment of odd and even loop boundaries). It took significant effort to accomplish execution time slightly below the one of the SSSRFB kernel at an execution rate of less than half of the peak.

There is less to be said about the two remaining kernels, SLARFB and SGEQRT. The SLARFB kernel turnes out to deliver very good performance without much effort. On the other hand, SGEQRT does not deliver good performance despite efforts similar to the STSQRT kernel. This is to be expected, however, since none of the loops have fixed boundaries. This kernel is executed the least and its poor performance does not affect the overall performance much. The situation is analogous to the SPOTRF kernel of the Cholesky factorization, for which similar performance is reported (roughly 6 Gflop/s [8]).

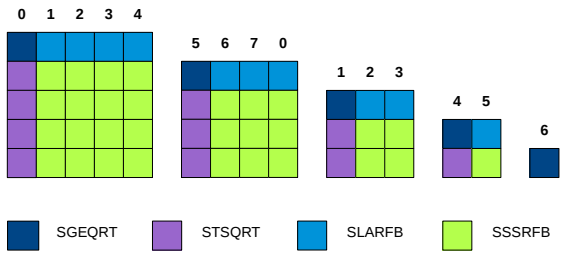


Figure 3: Cyclic partitioning of work to eight SPEs in the five consecutive steps of factorizing a 5×5 blocks matrix.

The last technical detail, which has not been revealed so far, is that the $T$ factors are stored in a compact format. Each element of $T$ is pre-splatted across a 4-element vector; each 4×4 triangular block of $T$ is stored in a column of 10 vectors and the $T$ array contains 16 such columns of overall size of 2560 bytes.

## 4.3 SPE Parallelization

For the distribution of work for parallel execution on the SPEs, static 1D cyclic partitioning is used, shown in Figure 3. The effect of "wrapping" the SPEs assignment from one step to another results in pipelining of factorization steps, basically implementing the technique known in linear algebra as the *lookahead*. Following Figure 3, one can observe that SPE 5 can start factorizing the second panel as soon as SPE 1 finishes the first SSSRFB operation.

Static work partitioning makes the synchronization extremely straightforward. With all the work predetermined, each SPE can proceed on its own, and only

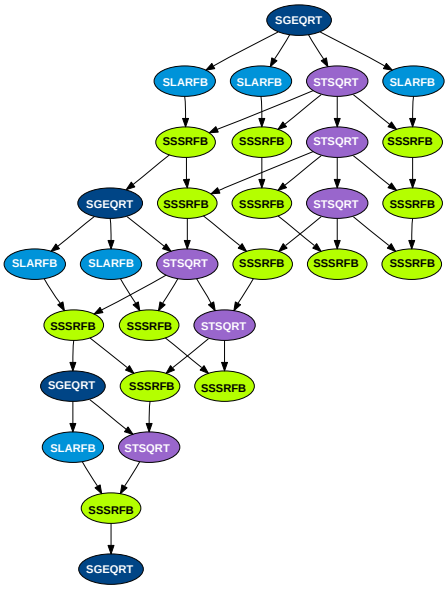


Figure 4: The DAG (Direct Acyclic Graph) of a tile QR factorization of a 4×4 blocks matrix.

needs to check if dependencies are satisfied for each operation. Figure 4 shows the dependencies between tasks of the tile QR algorithm expressed as a DAG (Direct Acyclic Graph).

Before fetching a tile for an operation in a given step, the SPE needs to check if the preceding step has completed on that tile. The SPE does that by looking 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 (byte) for each tile of the input matrix, indicating the number of the step which has completed on that tile. At the completion of an operation, an SPE broadcasts the progress information to all progress tables with an LS-to-LS DMA.

As one can see, the scheme implements the right-looking (aggressive) variant of the algorithm. Although different scenarios can be easily imagined, this version makes sense from the standpoint of ease of implementation. In this arrangement, an SPE factorizing the panel can hold the diagonal tile in place, while streaming the tiles below diagonal through Local Store. Similarly, an SPE updating a column of the trailing submatrix can hold the topmost tile in place, while streaming the tiles below it through Local Store. Data reuse is accomplished this way, which minimizes the traffic to main memory. It needs to be pointed out, though, that this is absolutely not necessary from the standpoint of memory bandwidth. The tile QR factorization is so compute intensive that all memory traffic can easily be hidden behind computation with data reuse or without it.

**Algorithm 1** Double buffering of communication in the tile QR implementation.

```
while more work to do do
    if data not prefetched then
        wait for dependencies
        fetch data
    end if
    if more work to follow then
        if dependencies met then
            prefetch data
        end if
    end if
    compute
    swap buffers
end while
```

At each step, the tiles of the input matrix are exchanged between the main memory and Local Store. Important aspect of the communication is double-buffering. Since work partitioning is static, upcoming operations can be anticipated and the necessary data fetched. In fact all data buffers are duplicated and, at each operation, a prefetch of data is initiated for the following operation (subject to dependency check). If the prefetch fails for dependency reasons, data is fetched in a blocking mode right before the operation. Algorithm 1 shows the mechanism of double buffering in the tile QR implementation.

Figure 5 shows the execution trace of factorizing a 512×512 matrix using all the eight SPEs.

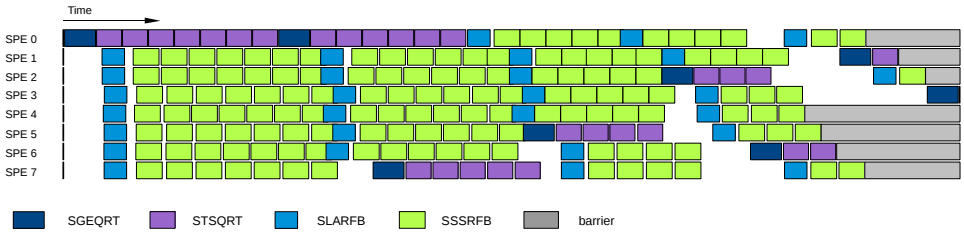


Figure 5: Execution trace of a factorization of a 512×512 matrix. (total time: 1645 $\mu$s, execution rate: 109 Gflop/s).

# 5 Results

Results presented in this section are produced on a single 3.2 GHz CELL processor of a QS20 dual-socket blade running Fedora Core 7 Linux and on a PlayStation 3 running Fedora Core 7 Linux. The code is cross-compiled using x86 SDK 3.0, although, as mentioned before, the kernels are cross-compiled with an old x86 SPU GCC 3.4.1 cross-compiler, since this compiler yields the highest performance. The results are checked for correctness by comparing the $R$ factor produced by the algorithm to the $R$ factor produced by a call to the LAPACK routine SGEQRF ran on the PPE.

It also needs to be mentioned that the implementation utilizes *Block Data Layout* (BDL), where each tile is stored in a continuous 16 KB portion of the main memory, which can be transferred in a single DMA, what puts an equal load on all 16 memory banks. Tiles are stored in the row-major order, and also data within tiles is arranged in the row-major order, a common practice on the CELL processor. Translation from standard, (FORTRAN) layout to BDL can be implemented very efficiently on the CELL processor [7]. 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 "faluted in" at initialization. As a result, an SPE never incurs a TLB miss during the run. Finally, on the QS20 blade it is assured that the memory is allocated on the NUMA node associated with the processor.

Table 3 and Figure 6 show the performance of the algorithm in Gflop/s, while using the standard formula, $2MN^2 -^2/_3 N^3$, for operation count. Table 3 also shows the percentage of the processor's peak of 204.8 Gflop/s and the percentage of the SSSRFB performance times the number of SPEs, which may serve as a quality measure for scheduling, synchronization and communication.

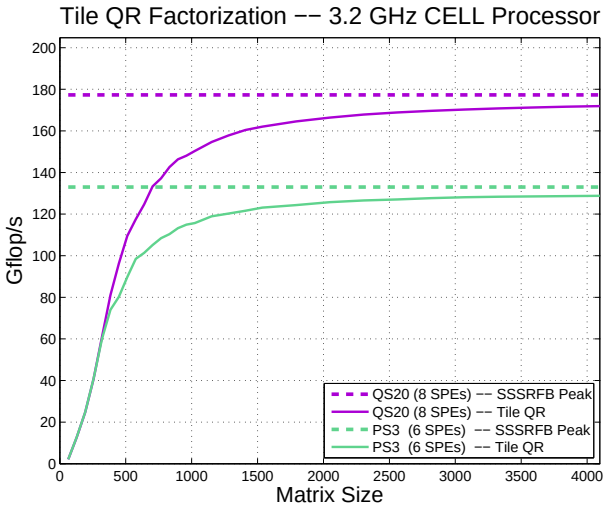


Figure 6: Performance of the tile QR factorization in single precision on a single CELL processor (8 SPEs) of a QS20 dual-socket blade and on a PlayStation 3 (6 SPEs). Square matrices were used. The dashed horizontal lines mark performance of the SSSRFB kernel times the number of SPEs.

Table 3: Performance of the tile QR factorization in single precision on a single 3.2 GHz CELL processor (8 SPEs) of a QS20 dual-socket blade. (Bold font indicates the point of exceeding half of the processor peak.)

| Matrix Size[a] | Execution Rate [Gflop/s][b] | Fraction of Peak [%][c] | Fraction of SSSRFB Peak [%][c] |
|---|---|---|---|
| 128 | 12 | 6 | 7 |
| 256 | 40 | 20 | 23 |
| 384 | 81 | 40 | 46 |
| **512** | **109** | **53** | **62** |
| 640 | 124 | 61 | 70 |
| 768 | 137 | 67 | 77 |
| 896 | 146 | 71 | 83 |
| 1024 | 150 | 73 | 85 |
| 1280 | 157 | 77 | 89 |
| 1536 | 162 | 79 | 91 |
| 2048 | 166 | 81 | 94 |
| 2560 | 168 | 82 | 95 |
| 3072 | 170 | 83 | 96 |
| 3584 | 171 | 84 | 97 |
| 4096 | 171 | 84 | 97 |

[a]square matrices were used

[b]values are truncated

[c]values are rounded

The presented implementation crosses half of the peak performance for problems as small as 512×512. For a 1024×1024 problem, it reaches 150 Gflop/s. It plateaus (gets close to its asymptotic performance) for problems larger than 2,500×2,500.

The code used to produce the reported results is freely available through one of the author's web site, http://www.cs.utk.edu/~kurzak/.

# 6 Conclusions

The presented implementation of tile QR factorization on the CELL processor allows for factorization of a 4000×4000 dense matrix in single precision in exactly half a second. To the authors' knowledge, at present, it is the fastest reported time of solving such problem by any semiconductor device implemented on a single semiconductor die.

It has been demonstrated that a complex dense linear algebra operation, such as the QR factorization, can be very efficiently implemented on a modern multi-core processor, such as the CELL processor, through the use of appropriate algorithmic approaches. Specifically, fine granularity of parallelization and loose model of synchronization allow for achieving high performance.

It has been shown that a short-vector SIMD architecture, such as the one of the SPE, can handle complex operations very efficiently, although, at this moment, significant programming effort by an experienced programmer is required.

# 7 Future Work

Based on experiences with Cholesky factorization, the authors have no doubt that the QR implementation can easily be extended to efficiently utilize two CELL processors in the QS20 blade.

Experiences with solutions of linear systems of equations using LU and Cholesky factorizations show that the technique of mixed-precision, iterative refinement can be used to achieve double precision accuracy, while exploiting the speed of single precision. It would be straightforward to apply the same approach to solve linear systems of equations or least squares problems using QR factorization. In fact, due to the higher cost, in terms of floating point operations of the QR factorization, the overhead of the iterative process will be much smaller than for the other cases.

Finally, it should be pointed out that LU factorization can be implemented in the same manner, yielding the tile LU algorithm that is bound to produce scaling similar to the QR and Cholesky algorithms, which is superior to the LU implementation reported so far. Although, it needs to be pointed out that the tile LU algorithm has different properties in terms of numerical stability.

## 8 Acknowledgements

The authors thank Alfredo Buttari and Julien Langou for their insightful comments, which helped immensely to improve the quality of this article.

## References

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

[2] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[3] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[5] Basic Linear Algebra Technical Forum. *Basic Linear Algebra Technical Forum Standard*, August 2001.

[6] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine architecture and its first implementation, A performance view. `http://www-128.ibm.com/developerworks/power/library/pa-cellperf/`, November 2005.

[7] 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.

[8] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving Systems of Linear Equation on the CELL Processor Using Cholesky Factorization. *Trans. Parallel Distrib. Syst.*, 2008.

[9] D. Hackenberg. Einsatz und Leistungsanalyse der Cell Broadband Engine. Institut für Technische Informatik, Fakultät Informatik, Technische Universität Dresden, February 2007. Großer Beleg.

[10] D. Hackenberg. Fast matrix multiplication on CELL systems. `http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/`, July 2007.

[11] W. Alvaro, J. Kurzak, and J. J. Dongarra. Fast and small short vector SIMD matrix multiplication kernels or the synergistic processing element of the CELL processor. In *2008 International Conference on Computational Science*. Lecture Notes in Computer Science 5101:935-944, 2008.

[12] S. Williams, L Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *ACM/IEEE SC'07 Conference*, 2007.

[13] A. Pothen and P. Raghavan. Distributed orthogonal factorization: givens and householder algorithms. *SIAM J. Sci. Stat. Comput.*, 10(6):1113–1134, 1989.

[14] E. Chu and A. George. QR factorization of a dense matrix on a hypercube multiprocessor. *SIAM J. Sci. Stat. Comput.*, 11(5):990–1028, 1990.

[15] M. W. Berry, J. J. Dongarra, and Y. Kim. LAPACK Working Note 68: A Highly Parallel Algorithm for the Reduction of a Nonsymmetric Matrix to Block Upper-Hessenberg Form. Technical Report UT-CS-94-221, Computer Science Department, University of Tennessee, 1994.

[16] P. E. Gill, G. H. Golub, W. A. Murray, and M. A. Saunders. Methods for Modifying Matrix Factorizations. *Mathematics of Computation*, 28(126):505–535, 1974.

[17] 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.

[18] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. LAPACK Working Note 191: A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Technical Report UT-CS-07-600, Electrical Engineering and Computer Science Department, University of Tennessee, 2007.

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

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