Neural, Parallel, and Scientific Computations 21 (2013) 111-128

EXPLOITING THE CAPABILITIES OF INTEL MULTI-CORE PROCESSORS FOR BLOCK QR DECOMPOSITION ALGORITHM

MOSTAFA I. SOLIMAN
Computer and System Section, Electrical Engineering Department
Faculty of Engineering, Aswan University, Aswan 81542, Egypt.
mossol@ieee.com and mossol@yahoo.com

Abstract. This paper shows how to make the QR decomposition algorithm run faster on Intel multi-core processors by exploiting explicit parallelism and memory hierarchy. Streaming SIMD extensions and multithreading computation on multiple cores are used to exploit data-level parallelism (DLP) and thread-level parallelism (TLP), respectively. In addition, memory hierarchy is exploited by performing the QR computation on blocks of data to reduce the impact of memory latency by reusing the loaded data in cache memories. On Core 2 Duo E7500 with two cores (2-physical/2-logical processors), Core i5 M520 with two cores supporting Hyper-Threading technology (2-physical/4-logical processors), and Xeon E5410 with four cores (4-physical/4-logical processors), the speedups of multithreaded SIMD implementations of the block QR decomposition on 20000×20000 dense matrices range 6.6-80, 11-103, and 6.6-80 times higher than the unparallel execution, respectively, when changing the block size from 4×4 to 100×100. Further increasing the block size to 500×500 reduces the speedups to 20, 34, and 19, respectively. Our results show performances of 80, 90, and 65 GFLOPS achieved on Core 2 Duo E7500, Core i5 M520, and Xeon E5410, respectively.

Keywords - DLP; Householder transformation; matrix blocking; memory hierarchy; multi-core processors; multithreading; QR decomposition; performance evaluation; SIMD; TLP.

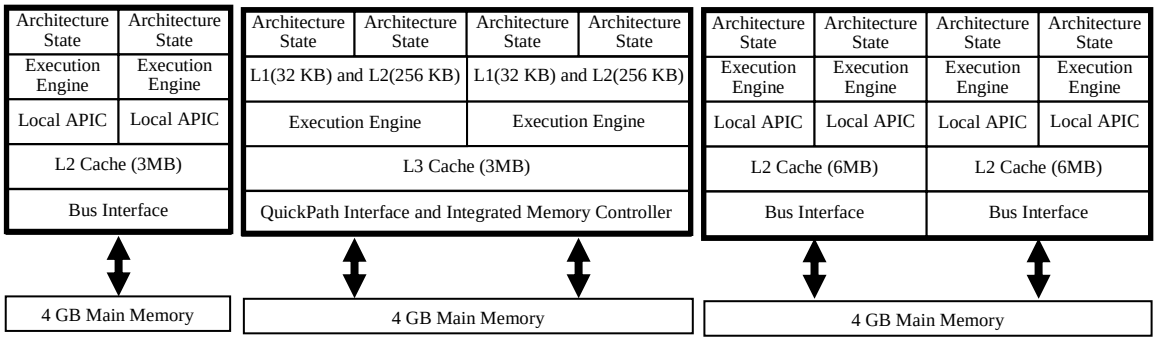
1. INTRODUCTION

Although transistor densities continue to improve according to Moore's Law [1], computer architects are finding it difficult to turn these transistors into single-thread performance. This is because the general-purpose single-threaded processors have become significantly limited by power consumption and they have reached the point of diminishing returns in performance. Thus, the industry has begun to de-emphasize single-thread performance and focus on integrating multiple cores onto a single die (multi-core processors) [2-7]. Moreover, the number of cores is expected to approximately double every two years [4]. According to Amdahl's Law [8], if programmers can parallelize their applications, then whole application performance should be doubled every two years and started again to track Moore's Law.

Received May 19, 2012 1061-5369 $15.00 © Dynamic Publishers, Inc

On multi-core processors, various forms of parallelism can be exploited to provide increases in performance above and beyond those made possible just by improvements in IC processing technology. Instruction-level parallelism (ILP), thread-level parallelism (TLP), and data-level parallelism (DLP) are the major ways in which processor designs can exploit parallelism to improve performance [9]. Whereas the compiler and hardware conspire to exploit ILP implicitly without the programmer's attention, TLP and DLP are explicitly parallel, requiring the programmer to write parallel code to gain performance. The common approach to exploit TLP is to decompose each parallel section into a set of tasks and distributes these tasks to different threads (see [10-12] for more detail). In addition to ILP and TLP, the use of multimedia extensions, such as Intel streaming SIMD extensions (SSE, SSE2, SSE3, and SSSE3 [13]), represents a good way to exploit DLP on a single processor core by processing multiple data using a single instruction. Thus, multi-core processors provide applications with an opportunity to achieve much higher performance than single-core processors. Furthermore, the number of cores on multi-core processors is likely to continue growing, increasing the performance potential of multi-core processors [4].

This paper shows how the performance of the QR decomposition algorithm can be improved on Intel multi-core processors through the exploitation of TLP, DLP, and memory hierarchy. Multithreading computation on multiple cores and streaming SIMD extensions are used to exploit TLP and DLP, respectively. In addition, memory hierarchy is exploited by performing the QR computation on blocks of data to reduce the impact of memory latency by reusing the loaded data in cache memories. The target systems are (1) Dell OptiPlex 380, which has Intel Core 2 Duo E7500 with two cores (2-physical/2-logical processors, see Figure 1a) running at 2.93 GHz and 4 GB memory, (2) HP ProBook 6550b, which has Intel Core i5 M520 with two cores supporting Hyper-Threading technology (2-physical/4-logical processors, see Figure 1b) running at 2.4 GHz and 4 GB memory, and (3) Fujitsu Siemens CELSIUS R550, which has Intel Xeon E5410 with four cores (4-physical/4-logical processors, see Figure 1c) running at 2.33 GHz and 4 GB memory. Table 1 illustrates the specifications of the Intel multi-core



(a) Intel Core™ 2 Duo E7500 (b) Intel Core™ i5 M520 (c) Intel Xeon® 5410

Figure 1: Target Intel multi-core processors.

Table 1: Specifications of the target multi-core processors.

Processor	Intel Core™ 2 Duo E7500	Intel Core™ i5 520M	Intel Xeon® E5410
# Cores	2	2	4
# Threads	2	4	4
Hyper-Threading	Not support	Support	Not support
Clock Speed	2.93 GHz	2.4 GHz	2.33 GHz
Cache	L2: 3 MB	L3: 3 MB	L2: 12 MB
Bus/Core Ratio	11	18	7
FSB Speed	1066 MHz	1066 MHz	1333 MHz
Instruction Set	64-bit	64-bit	64-bit
Lithography	45 nm	32 nm	45 nm
Processing Die Size	82 mm^2	81 mm^2	214 mm^2
# Transistors/Processing Die	228 million	382 million	820 million
Max TDP	65 W	35 W	80 W

processors used in our implementation of QR decomposition algorithm. C++ programming language has been used to code various implementations of QR decomposition algorithms using Microsoft Visual Studio 2008. Since C++ stores 2-D matrices in row major order in linear memory, row oriented QR decomposition algorithm is demonstrated in this paper.

The rest of the paper is organized as follows. Section 2 provides a brief discussion of related work. Section 3 presents the Householder QR performance improvements on Intel multi-core processors using Intel streaming SIMD extensions and multithreading. Section 4 describes the block QR algorithm and its multithreaded SIMD performance. Finally, the conclusion is drawn in Section 5.

2. RELATED WORK

In linear algebra, a QR decomposition (factorization) of a matrix is a decomposition of a matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R ($A = Q \times R$) [14]. QR decomposition is often used to solve the linear least squares problem, which has been used in many scientific fields such as mathematics, physics, statistics, and economics [15, 16]. Thus, in recent years, many researchers are working in the improvement of the parallel performance of the QR factorization algorithm to reduce its execution time.

Agullo et al. [17] tackled the issue of tuning a dense QR factorization on multi-core architectures using a fully empirical approach. Their method is automatic, fast, and reliable. They achieved an average performance varying from 97% to 100% of the optimum performance depending on the platform.

Dongarra et al. [18] described new QR factorization algorithm which is especially designed for massively parallel platforms combining parallel distributed multi-core nodes. Their new algorithm falls in the category of the tile algorithms [19-21] which naturally enables good data locality for the sequential kernels executed by the cores, low number of messages in a parallel distributed setting, and fine granularity. Their

experiments on the Grid'5000 edel platform showed the following gains at both ends of the matrix shape spectrum: (1) on tall and skinny matrices, they reached 57.5% of theoretical computational peak performance, to be compared with 6.4% for SCALAPACK (9.0x speedup) and (2) on square matrices, they reached 68.7% of theoretical computational peak performance, to be compared with 44.2% for SCALAPACK (1.6x).

Hadri et al. [22] presented a new fully asynchronous method for computing a QR factorization on shared memory multi-core architectures that overcame the limited performance when factorizing tall and skinny matrices or small square matrices. They adapted an existing algorithm that performs a panel factorization in parallel (named Communication-Avoiding QR and initially designed for distributed-memory machines), to the context of tile algorithms using asynchronous computations. Experimental study showed significant improvement (up to almost 10 times faster) compared to state of the art approaches.

Kurzak and Dongarra [23] presented implementation of tile QR factorization on the CELL processor (3.2 GHz CELL processor of a QS20 dual-socket blade) allowed for factorization of a 4000×4000 dense matrix in single precision in exactly half a second. They demonstrated how the potential of the CELL processor could be utilized to the fullest by employing the new algorithmic approach and successfully exploiting the capabilities of the CELL processor in terms of ILP and TLP.

Demmel et al. [24] presented parallel and sequential dense QR factorization algorithms that are both optimal in the amount of communication they perform, and just as stable as Householder QR. Their first algorithm, TSQR (Tall Skinny QR), factors $m \times n$ matrices in a 1-D block cyclic row layout, and is optimized for $m >> n$. The second algorithm, CAQR (Communication-Avoiding QR), factors general rectangular matrices distributed in a 2-D block cyclic layout. It invokes TSQR for each block column factorization. They implemented parallel TSQR on several machines, with speedups of up to 6.7× on 16 processors of a Pentium III cluster, and up to 4× on 32 processors of a BlueGene/L. They also modeled the performance of our parallel CAQR algorithm, yielding predicted speedups over ScaLAPACK's PDGEQRF of up to 9.7× on an IBM Power5, up to 22.9× on a model Petascale machine, and up to 5.3× on a model of the Grid.

Quintana-Orti et al. [25] examined the scalable parallel implementation of QR factorization of a general matrix, targeting SMP and multi-core architectures. They presented two implementations of algorithms-by-blocks, where each implementation viewed a block of a matrix as the fundamental unit of data, and likewise, operations over these blocks as the primary unit of computation. On SGI Altix 350 server (ccNUMA architecture with eight nodes, each with two 1.5 GHz Intel Itanium2 processors, providing a total of 16 CPUs and a peak performance of 96 GFLOPS), they achieved 73% of the peak performance. Moreover, on NEUMANN (SMP server with eight AMD

Opteron processors, each one with two cores at 2.2 GHz, providing a total of 16 cores and a peak performance of 70.4 GFLOPs), they achieved 65% of the peak performance.

This paper shows how to make the QR decomposition algorithm run faster on Intel multi-core processors by exploiting explicit parallelism (TLP and DLP) and memory hierarchy.

3. HOUSEHOLDER QR DECOMPOSITION ON INTEL MULTI-CORE PROCESSORS

Given an $m \times n$ real matrix A, with $m \geq n$, its QR factorization is given by $A = Q \times R$, where the $m \times m$ matrix Q is orthogonal ($Q^T Q = QQ^T = I$), and the $m \times n$ matrix R is upper triangular. There are many different methods for computing the QR factorization, where the standard algorithm involves successive Householder transformations (see [14] for more detail).

Householder transformations (reflectors) are rank-1 modifications of the identity and they can be used to zero selected elements of a vector. Suppose that x is n-element *row* vector and xP is needed to be a multiple of $e1 = \{1, 0, 0, \ldots, 0\}$. Matrix P is called Householder transformation and is calculated as follows:

$$P = I - (2v^T v / vv^T),$$

where the vector v is called Householder vector and xP is reflected in the hyperplane span $v^{\perp}$. Note that $v^T v$ results in outer-product and vv^T produces dot-product. Moreover, Householder matrix P is symmetric and orthogonal. P is used to annihilate selected elements of vector x as follows:

$$xP = x(I - 2v^T v / vv^T) = x - (2xv^T / vv^T)v.$$

Let $v = x + \alpha e1$, then

$$xv^T = xx^T + \alpha x_1 \text{ and } vv^T = xx^T + 2\alpha x_1 + \alpha^2.$$

Thus, in order to make coefficient of x to be zero, set

$$\alpha = \pm ||x||_2.$$

Therefore, if $v = x - ||x||_2 e_1$, then

$$xP = +||x||_2 e_1.$$

See [14] for more detail.

In Listing 1, lines 2-14 show the calculation of Householder vector v and its scalar value $b = (2/vv^T)$ of row i of an $n \times n$ matrix A ($A[i][1:i\text{-}1]$), where i varies from n to 1. It is clear that the calculation of Householder vector is based on Level-1 BLAS [26]. It involves about $3n$ floating-point operations (FLOPs), mainly for dot-product ($2n$ FLOPs) in line 2 and vector scaling (n FLOPs) in line 14, where n is the vector length. Moreover, applying Householder reflection (see Listing 1, lines 15-17) to the rows of matrix A ($A \times P$) is based on Level-2 BLAS [27]. It involves a matrix-vector multiplication ($2n^2$

Listing 1: QR factorization using Householder reflector.

```
01  FOR i = n to 1 step -1
02    s = A[i][1:i-1]×A[i][1:i-1]^T
03    v[i] = 1
04    v[1:i-1] = A[i][1:i-1]
05    IF( s = 0 )
06      b = 0
07    ELSE
08      u = sqrt(A[i][i]^2 + s )
09      IF(A[i][i] < 0)
10        v[i] = A[i][i] - u
11      ELSE
12        v[i] = -s/( A[i][i] + u )
13      b = 2*v[i]^2/( v[i]^2 + s )
14      v[1:i] = v[1:i]/v[i]
15    FOR  j = i to 1 step -1
16      w[j]^T = b*(A[j][1:i]×v[1:i]^T)
17      A[j][1:i] = A[j][1:i] - w[j]^T*v[1:i]
```

FLOPs) and outer-product update ($2n^2$ FLOPs) by exploiting the structure of Householder reflector, where the matrix size is $n \times n$, as follows:

$$A \times P = A(I - 2v^{T}v/vv^{T}) = A - 2Av^{T}v/vv^{T} = A - w^{T}v,$$

where, $w^{T} = bAv^{T}$. In Listing 1, lines 1-17 show how Householder transformation is used to perform QR factorization on $n \times n$ matrix A. Lines 2-14 calculate Householder vector v and its scalar value b, while lines 15-17 apply v on the remaining rows.

The total number of FLOPs needed for QR factorization shown in Listing 1 on an $n \times n$ matrix is $4n^3/3$ [14]. The execution time for applying Householder reflection dominates the overall time for calculating the QR factorization (see Figure 2). On reasonably large matrix size, accelerating the calculation of Householder vector v and its scalar value b (Level-1 BLAS) improves the performance of QR decomposition with negligible value (1%) by applying Amdahl's Law [8]. This means, improving the performance of QR decomposition requires reducing the execution time of applying Householder reflections (Level-2 BLAS) using parallel processing techniques. However, the performance of Level-2 BLAS is limited by the memory access bandwidth [28]. It involves $O(n^2)$

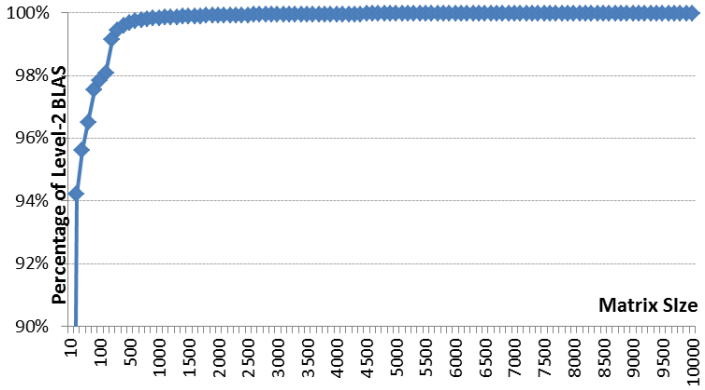
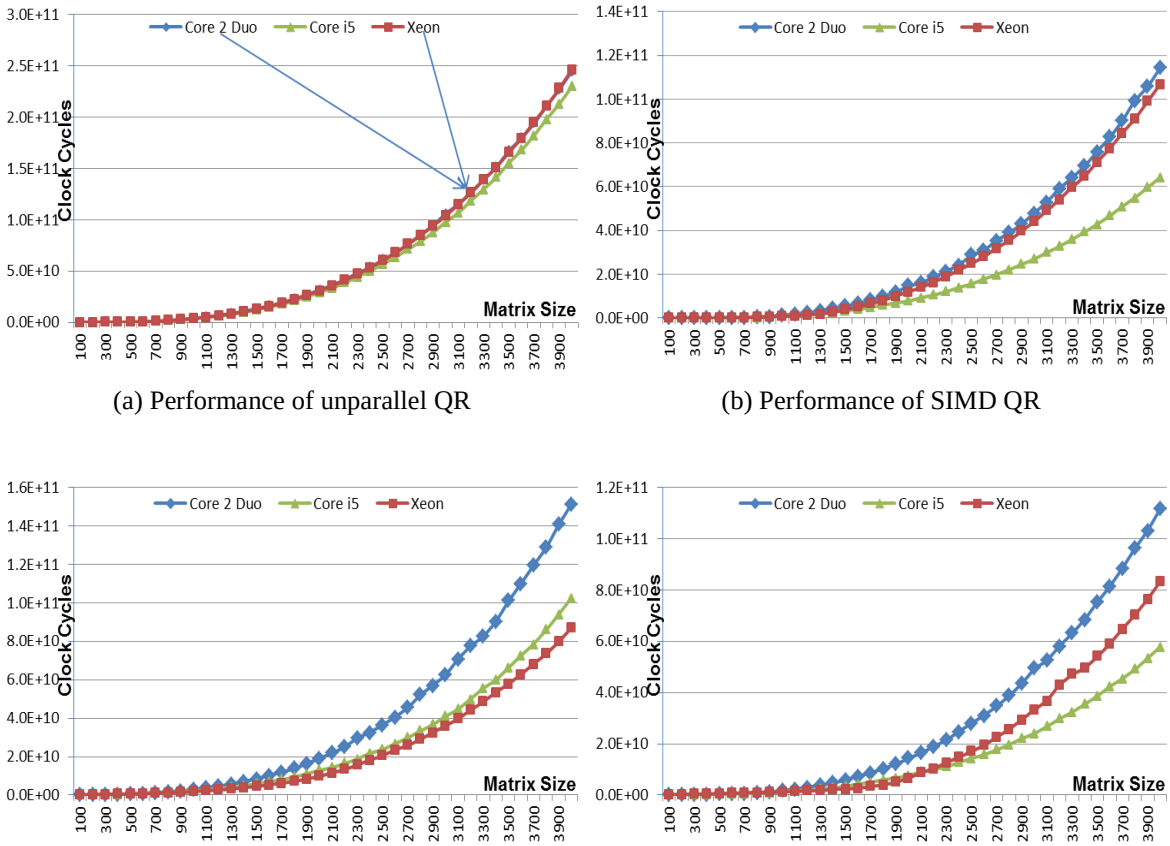


Figure 2: Percentage of Level-2 BLAS in QR decomposition Algorithm.

arithmetic operations on $O(n^2)$ data, where the dimension of the matrix involved is $n \times n$. The ratio of arithmetic operations to memory references is $O(1)$, which is not enough to improve the performance of multi-core processors with a hierarchy of memory because of the low-level of reusing data.

Figure 3a shows the performance in clock cycles of unparallel QR decomposition illustrated in Listing 1 on Intel multi-core processors shown in Figure 1. On 1000×1000 matrices, the execution times are $(3.83*10^9)/(2.93*10^9)$, $(3.56*10^9)/(2.4*10^9)$, and $(3.75*10^9)/(2.33*10^9)$, or 1.3, 1.5, and 1.6 seconds on Core 2 Duo, Core i5, and Xeon processors, respectively. Moreover, on matrices of size 4000×4000, the execution times are $(2.45*10^{11})/(2.93*10^9)$, $(2.3*10^{11})/(2.4*10^9)$, and $(2.46*10^{11})/(2.33*10^9)$, or 83.6, 95.8, and 105.6 seconds, respectively. This means, on the unparallel version of QR factorization shown in Listing 1, the performance of Core 2 Duo is the best because it has the highest running frequency (2.93 GHz) and only one logical processor is exploited. For the same reason, the performance of Core i5 is better than Xeon even though the processing die size of the Xeon processor is the largest (see Table 1).

On large matrices, the execution time is unacceptable (more than 3 hours on 20000×20000). Thus, parallel processing techniques are used for improving the performance (reducing the execution time) of QR decomposition. Figure 3b shows that



(a) Performance of unparallel QR

(b) Performance of SIMD QR

(c) Performance of multithreaded QR

(d) Performance of multithreaded SIMD QR

Figure 3: Performance evaluation of Householder QR decomposition on Intel multi-core processors.

the use of Intel streaming SIMD extensions by processing four single-precision floating-point data using a single instruction reduces the execution QR decomposition. In addition, the speedups due to SIMD on various sizes of matrices are listed in Table 2a. The performance of Core i5 is the best because its memory hierarchy has three levels of cache memory.

Moreover, the effect of using multithreading technique is shown in Figure 3c and its speedup is listed in Table 2b. As expected, the performance of multithreaded QR on Xeon is the best since it has four execution cores, however, Core 2 Due and Core i5 each has two execution cores. In total, Figure 3d shows the effect of using both SIMD and multithreading techniques on the performance of QR decomposition. As Table 2c lists and Figure 3d shows, the best performance is on Core i5 because its memory hierarchy system supports the memory requirement of SIMD technique Moreover, Hyper-Threading technology enables two threads to run on each core (two logical processors) of Core i5. Excluding Core i5, the performance on Xeon is better than on Core 2 Duo because the former has greater number of cores than the later. Further improvements on performance can be achieved by switching to the block QR decomposition algorithm to exploit memory hierarchy in addition to SIMD and multithreading as the following section shows.

Table 2: Speedup due to parallel processing serial QR.

(a) Speedup due to SIMD

Matrix Size	Core 2 Duo E7500	Core™ i5 M520	Xeon E5410
500	4.43	5.00	4.48
1000	3.14	4.51	4.74
2000	2.12	3.66	2.60
4000	2.14	3.58	2.31
Average	**3**	**4.2**	**3.5**

(b) Speedup due to multithreading

Matrix Size	Core 2 Duo E7500	Core™ i5 M520	Xeon E5410
500	0.81	0.88	0.83
1000	1.4	1.67	2.09
2000	1.67	2.24	3.15
4000	1.62	2.25	2.84
Average	**1.4**	**1.8**	**2.2**

(c) Speedup due to multithreaded SIMD

Matrix Size	Core 2 Duo E7500	Core™ i5 M520	Xeon E5410
500	1.21	1.13	0.88
1000	2.31	2.88	3.02
2000	2.19	3.82	4.66
4000	2.19	3.98	2.96
Average	**2**	**3**	**2.9**

4. BLOCK HOUSEHOLDER QR DECOMPOSITION ON INTEL MULTI-CORE PROCESSORS

The gap between CPU speed and memory speed is increasing rapidly as new generations of computer systems are introduced [9]. To address this memory access bottleneck, most modern computers use multi-level memory hierarchies in their architectures. The key to improve the performance of applications on multi-level memory hierarchies is to avoid unnecessary memory references as well as to exploit locality by reusing the loaded data into a higher-level cache. Since the movement of data between memory and registers can have the same (or even more) cost as arithmetic operations, an algorithm performance can be dominated by the amount of memory traffic rather than by the number of arithmetic operations involved (like QR algorithm shown in Listing 1). This provides a considerable motivation to restructure existing algorithms and to devise new algorithms that minimize data movement by reusing the loaded data into cache memories [29]. A number of researchers have demonstrated the effectiveness of block algorithms on a variety of modern computer architectures (see [30] for more detail).

On multi-core Intel processors [13], the given problem can be partitioned into blocks. Parallel proceeding of these blocks are performed on multiple cores using multithreading technique to reduce the total execution time. Besides, Intel SIMD instructions can be used to perform operations on a block in parallel to further improve the performance. This approach provides for full reuse of data while the block is held in cache memory. It avoids excessive movement of data to/from main memory and gives a surface-to-volume effect for the ratio of arithmetic operations to data movement, i.e., $O(n^3)$ arithmetic operations to $O(n^2)$ data movement [31].

By reorganizing the computations of Householder QR decomposition shown in Listing 1, which is rich in the Level-2 BLAS [27], block Householder QR decomposition based on Level-3 BLAS [32] is constructed as shown in Listing 2. A product $Q = P_1 P_2 \ldots P_r$ of $n \times n$ Householder matrices can be written in the form $Q = I + W^T Y$, where W and Y are each $r \times m$ matrices (see [33, 34] for more detail).

$$AQ = AP_1P_2 \ldots P_r =$$

$$A(I - 2v_1^T v_1/v_1v_1^T)(I - 2v_2^T v_2/v_2v_2^T) \ldots\ldots (I - 2v_r^T v_r/v_rv_r^T) = A(I + W^T Y) = A + AW^T Y,$$

where, W and Y matrices can be calculated from Householder vectors $v_1, v_2, \ldots, v_r$ and their scalar values $b_1, b_2, \ldots, b_r$, as follows:

```
Y = v1                  // first row of Y is v1
W = –b1v1               // first row of W is scalar –b1 times v1
FOR  j = 2 to r step 1
   z = –bj(I + W^T Y)vj
   W = [W z]            // concatenate row z after W rows
   Y = [Y vj]           // concatenate row vj after Y rows
```

Listing 2: Block Householder QR decomposition.

```
01   FOR i = n to 1 step -b
02     FOR p = 0 to b-1 step 1
03       s = A[i-p][1:i-p-1]×A[i-p][1:i-p-1]^T
04       v[i-p] = 1
05       v[1:i-p-1] = A[i-p][1:i-p-1]
06       IF(s = 0)
07         b = 0
08       ELSE
09         u = sqrt(A[i-p][i-p]^2 + s )
10         IF(A[i-p][i-p] < 0)
11           v[i-p] = A[i-p][i-p] - u
12         ELSE
13           v[i-p] = -s/( A[i-p][i-p] + u )
14         b = 2*v[i-p]^2/( v[i-p]^2 + s )
15         v[1:i-p] = v[1:i-p]/v[i-p]
16       FOR j = i-p to i-b-1 step -1
17         w = b*(A[j][1:i-p]×v[1:i-p]^T)
18         A[j][1:i-p]=A[j][1:i-p] - w*v[1:i-p]
19       IF(p = 0)
20         Y[1][1:i] = v[1:i]
21         W[1][1:i] = -b * v[1:i]
22       ELSE
23         FOR j = 1 to p-1 step 1
24           y[j] = Y[j][1:i]×v[1:i]^T
25         W[p+1][1:i] = -b*(W[1:j][1:i]^T×y[1:j]+v[1:i])
26         Y[p+1][1:i] = v[1:i]
27     T[1:i-b][1:b]=A[1:i-b][1:i]×W[1:b][1:i]^T
28     A[1:i-b][1:i]+=T[1:i-b][1:b]×Y[1:b][1:i]
```

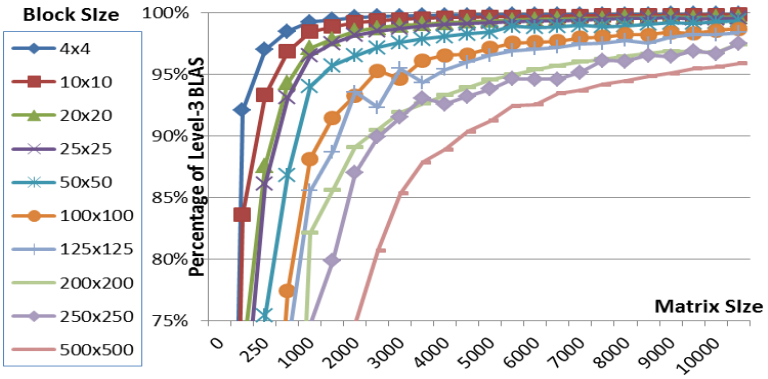


Figure 4: Percentage of Level-3 BLAS in block QR.

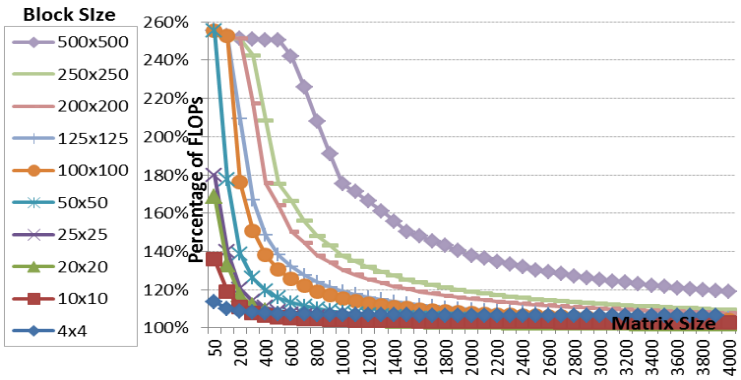
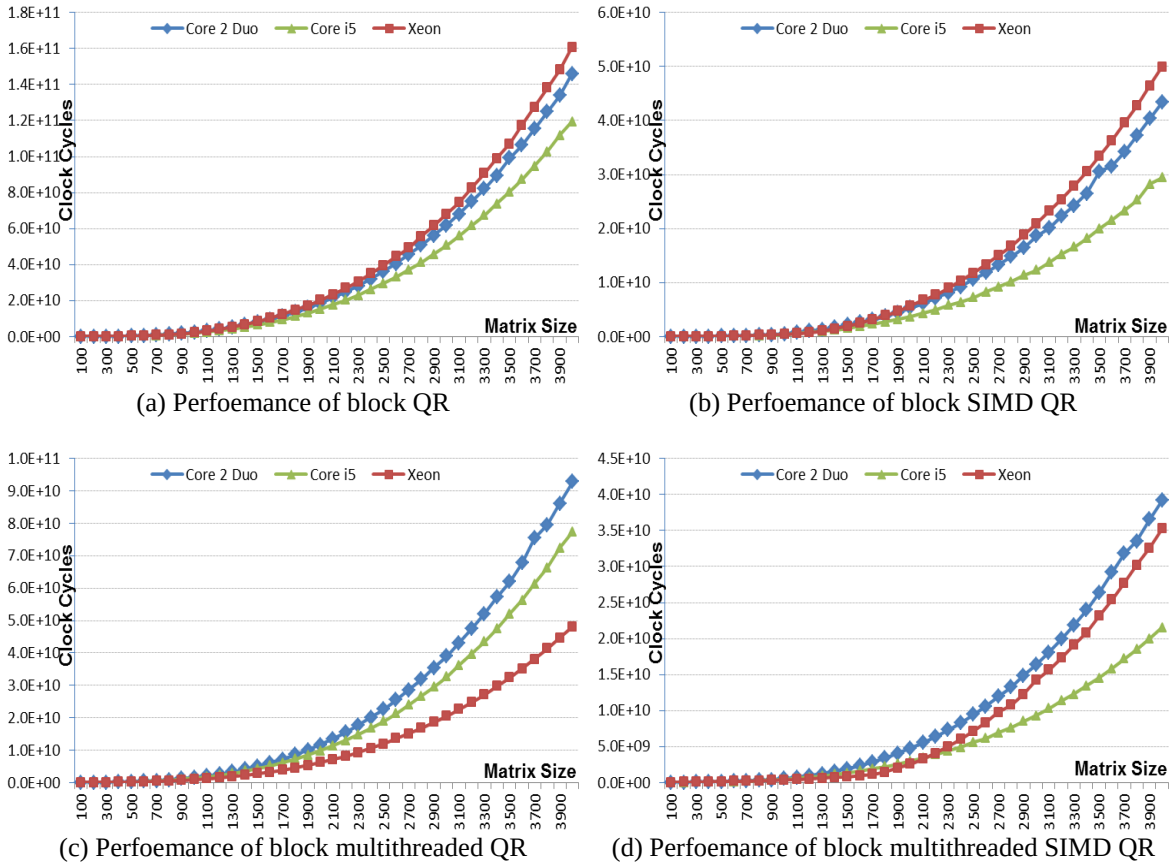


Figure 5: Increasing the number of FLOPs in the block QR.

Figure 4 shows the percentage of Level-3 BLAS in the block QR decomposition for multiple block sizes (4×4, 10×10, 20×20, 25×25, 50×50, 100×100, 125×125, 200×200, 250×250, and 500×500). It is clear that as the block size increases the percentage of Level-3 BLAS decreases because of the overhead of constructing W and Y matrices. Thus, the performance of the block QR decomposition is better on larger block size when the problem size is reasonably large, as we shall discuss. Moreover, increasing the block size increases the number of FLOPs needed for block QR decomposition. Figure 5 shows the percentage of increasing the number of FLOPs over the unblocked version (see Listing 1), which requires ($4n^3/3$ FLOPs). On 1000×1000 (4000×4000), the block QR decomposition needs 104%, 115%, 130%, and 175% (102%, 104%, 108%, and 119%) FLOPs when using blocks of sizes 25×25, 100×100, 200×200, and 500×500, respectively. This means, increasing the problem size or decreasing the block size or both results in decreasing the number of FLOPs of block QR factorization algorithm.

Figure 6a shows the performance in clock cycles of block QR decomposition illustrated in Listing 2 on Intel processors, where the block size is 4×4. The use of blocking technique only speeds up the execution of Householder QR decomposition by factors of 1.7, 1.9, 1.6 on Core 2 Duo, Core i5, and Xeon processors, respectively (see Table 3a). Furthermore, processing 4×4 blocks of QR using SIMD improves the performance, as shown in Figure 6b. On 4000×4000 matrices, the execution times on



(a) Perfoemance of block QR

(b) Perfoemance of block SIMD QR

(c) Perfoemance of block multithreaded QR

(d) Perfoemance of block multithreaded SIMD QR

Figure 6: Performance evaluation of QR based on block Householder transformation.

Core 2 Duo, Core i5, and Xeon are $(4.34*10^{10})/(2.93*10^{9})$, $(2.94*10^{10})/(2.4*10^{9})$, and $(5*10^{10})/(2.33*10^{9})$, or 14.8, 12.25, and 21.5 seconds. This means that the performance of Core i5 is the best because of the exploitation of three levels of memory hierarchy for reusing the loaded blocks. Table 3b shows the average speedup due to the use of blocking SIMD (6.5, 8.2, and 6.5, respectively).

Processing multiple blocks of QR decomposition algorithm on multiple execution cores using multithreading technique improves the performance furthermore (see Figure 6c). The average speedup on Xeon is the best (four times) since it has four execution cores (see Table 3c). Moreover, the speedup on Core i5 is better than Core 2 Due because its cores supports Hyper-Threading technology (each core can process two threads in parallel, see Table 1). In total, Figure 6d shows the effect of using blocking, SIMD, and multithreading techniques on the performance of QR decomposition. As Figure 7 and Table 3d show, on small matrix size, the speedup due to using blocking, SIMD, and multithreading techniques is almost equal on the three types of Intel multi-core processors. On large matrix size fitted in cache memory, the speedup on Xeon is the best because it has the largest number of physical cores. When the matrix size cannot fit in

Table 3: Speedup due to parallel processing block QR.

(a) Speedup due to blocking

Matrix Size	Core 2 Duo E7500	Core™ i5 M520	Xeon E5410
500	1.81	2.03	1.82
1000	1.61	1.88	1.67
2000	1.69	1.9	1.53
4000	1.68	1.92	1.53
Average	**1.7**	**1.9**	**1.6**

(b) Speedup due to block SIMD

Matrix Size	Core 2 Duo E7500	Core™ i5 M520	Xeon E5410
500	7.3	8.11	7.58
1000	7.23	8.87	8.25
2000	5.96	7.91	5.43
4000	5.64	7.81	4.93
Average	**6.5**	**8.2**	**6.5**

(c) Speedup due to block multithreading

Matrix Size	Core 2 Duo E7500	Core™ i5 M520	Xeon E5410
500	1.71	1.85	2.21
1000	2.45	2.59	3.85
2000	2.71	2.94	4.99
4000	2.63	2.97	5.12
Average	**2.4**	**2.6**	**4**

(d) Speedup due to block multithreaded SIMD

Matrix Size	Core 2 Duo E7500	Core™ i5 M520	Xeon E5410
500	3.37	3.72	3.31
1000	6.94	7.88	10.1
2000	6.59	9.64	12.1
4000	6.25	10.7	6.99
Average	**5.8**	**8**	**8.1**

cache memory, the speedup on Core i5 is the best since it has the three levels of memory hierarchy.

Figure 7 shows the effect of increasing block size from 4×4 to 10×10, 20×20, 25×25, 50×50, 100×100, 125×125, 200×200, 250×250, and 500×500 on the multithreaded SIMD implementation of the block QR decomposition. The performance is measured in GFLOPS (Giga floating-point operations per second) and is calculated as dividing the number of FLOPs of the main QR factorization algorithm ($4n^3/3$ for unblocked version in Listing 1) by the execution time in seconds (number of clock cycles times clock cycle time). The best performance can be achieved when the percentage of Level-3 BLAS is 95% or more with a largest block size because increasing the block size leads to decreasing the percentage of Level-3 BLAS (see Figure 4). Figures 8a, 8c, and 8e show performances of about performances of 80, 90, and 65 GFLOPS achieved on Core 2 Duo E7500, Core i5 M520, and Xeon E5410, respectively. On a fixed problem size, increasing the block size decreases the percentage of Level-3 BLAS and increases the numbers of FLOPs. However, increasing the block size increases the locality of data and decreases the execution time (number of clock cycles). Thus these tradeoffs results in turnover points as increasing the block sizes and fixing the problem size. The locations of these turnover points change with increasing the problem size.

Figures 8b, 8d, and 8f show the speedup of the multithreaded SIMD implementation of block QR factorization over the unparallel implementation of the unblocked version. Speedups of multithreaded SIMD implementation of the block QR decomposition on 20000×20000 dense matrix range 6.6-80, 11-103, and 6.6-80 times higher than the unparallel execution on Core 2 Duo E7500, Core i5 M520, and Xeon E5410, respectively, on block size varies from 4×4 to 100×100.

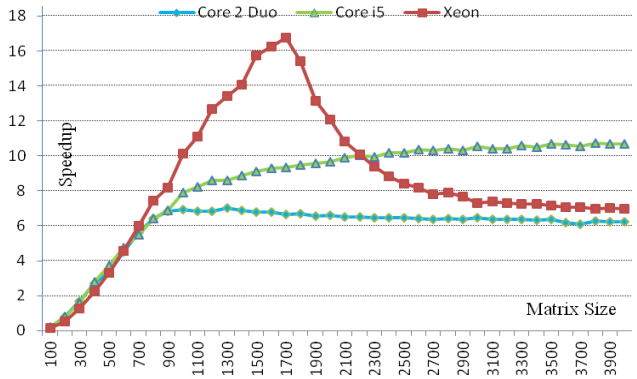
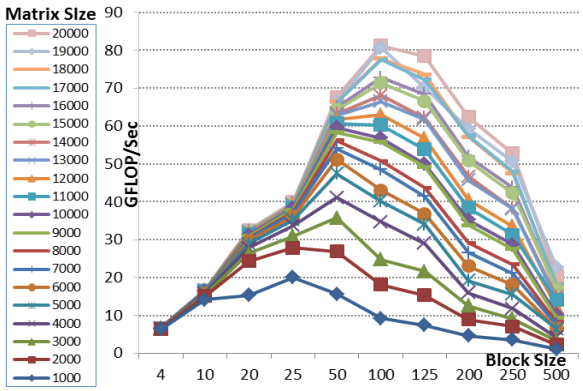
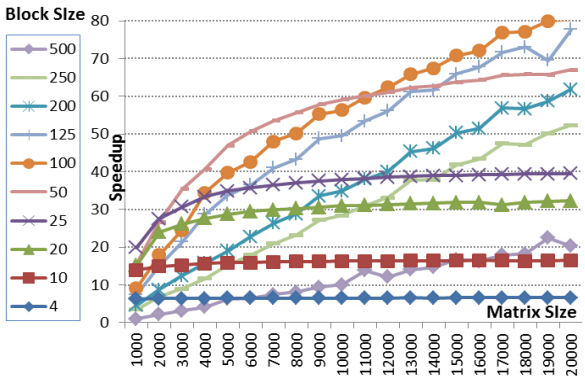


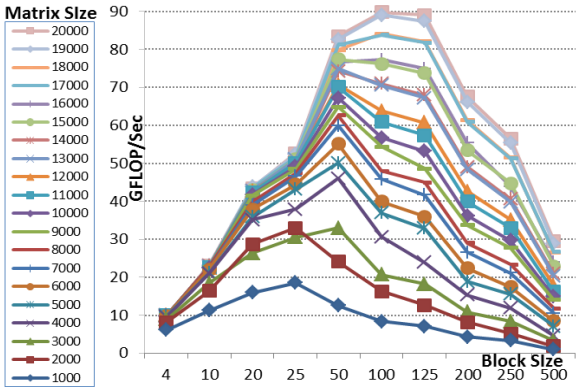
Figure 7: Speedup due to block multithreaded SIMD QR on small matrices.



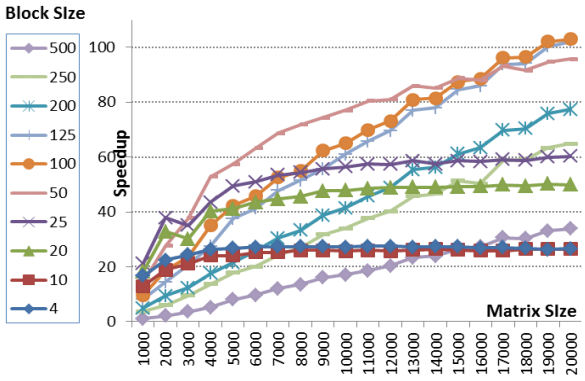
(a) Performance of Block QR on Core 2 Due



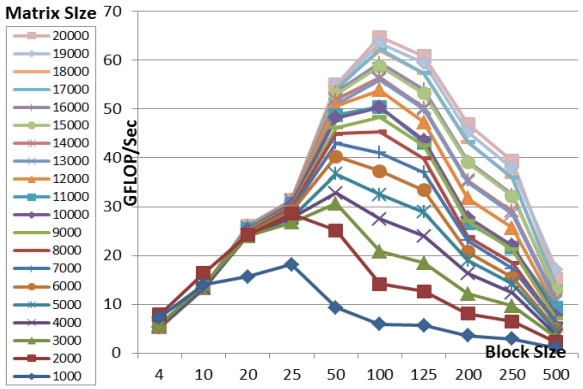
(b) Speedup on Core 2 Due



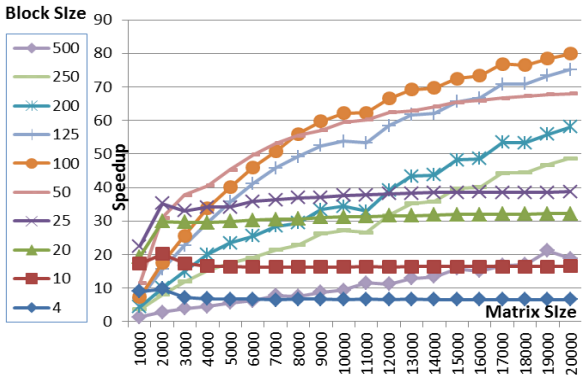
(c) Performance of Block QR on Core i5



(d) Speedup on Core i5



(e) Performance of Block QR on Xeon



(f) Speedup on Xeon

Figure 8: The effect of changing block size on the performance of the block Householder QR decomposition.

5. CONCLUSION

A combination of using efficient algorithms and well designed implementations leads to great high performance applications. This paper shows how the performance of the QR decomposition is enhanced through the exploitation of data-level parallelism (DLP), thread-level parallelism (TLP), and memory hierarchy on Intel multi-core processors. Streaming SIMD extensions and multithreading computation are used to exploit DLP and TLP, respectively. To facilitate the exploitation of TLP on multiple execution cores, the block QR algorithm is used.

On Core 2 Duo E7500 with two cores, Core i5 M520 with two cores supporting Hyper-Threading technology, and Xeon E5410 with four cores, the average speedup of block multithreaded SIMD implementation of the 4×4 block QR decomposition on matrices of small sizes 500×500 up to 1500×1500 in step of 100 are about 6.2, 7.1, and 9.7 times higher than the unparallel execution, respectively. As the matrices sizes increase (1500×1500 up to 4000×4000 in step of 100), the average speedup of 4×4 blocked multithreaded SIMD implementation are about 6.4, 10.2, and 9.5 times on Core 2 Duo E7500, Core i5 M520, and Xeon E5410, respectively. However, speedup of multithreaded SIMD implementation of the 4×4 block QR decomposition on 20000×20000 dense matrix range 6.6, 11, and 6.6 times higher than the unparallel execution on Core 2 Duo E7500, Core i5 M520, and Xeon E5410, respectively. Increasing the block size from 4×4 to 100×100 leads to improving the multithreaded SIMD implementation by 80, 103, and 80 times higher than the unparallel execution, respectively. Further increasing the block size to 500×500 reduces the speedups to 20, 34, and 19, respectively. Our results show performances of 80, 90, and 65 GFLOPS achieved on Core 2 Duo E7500, Core i5 M520, and Xeon E5410, respectively.

REFERENCES

[1] G. Moore, “Cramming More Components onto Integrated Circuits,” Electronics, Vol. 38, No. 8, pp. 114-117, April, 1965.

[2] K. Olukotun and L. Hammond, “The Future of Microprocessors,” ACM Queue, Vol. 3, No. 7, pp.26-29, September 2005.

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley,” Technical Report, EECS Department, University of California, Berkeley, December 2006.

[4] J. Shalf, J. Bashor, D. Patterson, K. Asanovic, K. Yelick, K. Keutzer, and T. Mattson, “The Manycore Revolution: Will HPC Lead or Follow?,” Journal of SciDAC Review, No. 14, pp. 40–49, Fall 2009.

[5] R. Buchty, V. Heuveline, W. Karl, and J. Weiss, "A Survey on Hardware-aware and Heterogeneous Computing on Multicore Processors and Accelerators," Karlsruhe Institute of Technology (KIT), Preprint Series of the Engineering Mathematics and Computing Lab (EMCL), ISSN 2191–0693 , No. 2009-02, Germany, 2009.

[6] G. Blake, R. Dreslinski, and T. Mudge, "A Survey of Multicore Processors," IEEE Signal Processing Magazine, vol. 26, issue 6, pp. 26-37, 2009.

[7] Dr. Dobb's, "The Parallel Programming Landscape: Multicore has gone mainstream--but are developers ready?," http://software.intel.com/sites/billboard/sites/default/files/PDFs/TW_1111059_StOfParallelProg_v6.pdf, 2012.

[8] G. Amdahl, "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities," Proc. AFIPS 1967 Spring Joint Computer Conference, Atlantic City, New Jersey, AFIPS Press, Vol. 30, pp. 483-485, April 1967.

[9] J. Hennessay, and D. Patterson, Computer Architecture A Quantitative Approach, 5th Edition, Morgan-Kaufmann, ISBN: 9780123838728, 2011.

[10] A. Binstock and R. Gerber, Programming with Hyper-Threading Technology: How to Write Multithreaded Software For Intel IA-32 Processors, Intel PRESS, ISBN 0970284691, 2003.

[11] R. Gerber, A. Bik, K. Smith, and X. Tian, The Software Optimization Cookbook: High-Performance Recipes for IA-32 Platforms, Second Edition, Intel PRESS, ISBN 0976483211, 2006.

[12] S. Akhter and J. Roberts, Multi-Core Programming: Increasing Performance through Software Multithreading, Intel PRESS, ISBN 0976483246, 2006.

[13] Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C,
Available at: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, Order Number: 325462-042US, March 2012.

[14] G. Golub and C. Van Loan, Matrix Computations, 3rd Edition, The Johns Hopkins University Press, Baltimore and London, 1996.

[15] E. Anderson, Z. Bai, and J. Dongarra, "Generalized QR Factorization and Its Applications," Linear Algebra and Its Applications, pp. 243-271, 1992.

[16] F. Song, H. Ltaief, B. Hadri, and Jack Dongarra, "Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems," Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11, 2010.

[17] E. Agullo, J. Dongarra, R. Nath, and Stanimire Tomov, "A Fully Empirical Autotuned Dense QR Factorization for Multicore Architectures," Proc. 17th International Conference on Parallel Processing, Volume Part II, Springer-Verlag Berlin, Heidelberg, 2011.

[18] J. Dongarra, M. Faverge, T. Herault, J. Langou and Y. Robert, "Hierarchical QR Factorization Algorithms for Multi-core Cluster Systems," University of Tennessee Computer Science Technical Report, UT-CS-11-684, October 2011.
Available at: http://www.netlib.org/lapack/lawns/.

[19] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel Tiled QR Factorization for Multicore Architectures," Concurrency Computation: Practice and Experience, Vol. 20, No. 13, pp. 1573-1590, September 2008.

[20] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures," Parallel Computing, Vol. 35, No. 1, pp. 38-53, January 2009.

[21] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert, "Tiled QR Factorization Algorithms," Proc. IEEE/ACM International Conference on High Performance Computing Networking, Storage and Analysis, SC'2011, 2011.

[22] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra, "Tile QR Factorization with Parallel Panel Processing for Multicore Architectures," Proc. IEEE International Symposium on Digital Object Identifier, Parallel and Distributed Processing (IPDPS), pp. 1-10, 2010.

[23] J. Kurzak, J. Dongarra, "QR Factorization for the CELL Processor," Scientific Programming," Vol. 17, No. 1-2, pp. 31-42, 2010.

[24] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-Optimal Parallel and Sequential QR and LU Factorizations," Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report No. UCB/EECS-2008-89, 2008. Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-89.html.

[25] G. Quintana-Ort´i, E. Quintana-Ort´ı, R. Geijn, F. Zee, and E. Chan, "Programming Matrix Algorithms-by-Blocks for Thread Level Parallelism," ACM Transactions on Mathematical Software, Vol. 36, No. 3, pp. 1-26, 2009.

[26] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," ACM Transactions on Mathematical Software, Vol. 5, No. 3, pp. 308-323, September 1979.

[27] J. Dongarra, J. Croz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," ACM Transactions on Mathematical Software, Vol. 14, No. 1, pp. 1-17, March 1988.

[28] M. Soliman, "Performance Evaluation of Multi-core Intel Xeon Processors on Basic Linear Algebra Subprograms," Parallel Processing Letters, Vol. 19, No. 1, Pages 159-174, March 2009.

[29] O. Brewer, J. Dongarra, and D. Sorensen, "Tools to Aid in the Analysis of Memory Access Patterns for FORTRAN Programs," Parallel Computing, Vol. 9, No. 1, pp. 25-35, December 1988.

[30] J. Demmel, J. Dongarra, J. Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, "Prospectus for the Development of a Linear Algebra Library for High-Performance Computers," Argonne National Laboratory Report, ANL-MCS-TM-97, Mathematics and Computer Science Division, September 1987.

[31] J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, L. Torczon, and W. Gropp, The Sourcebook of Parallel Computing, Morgan Kaufmann, ISBN 1558608710, November 2002.

[32] J. Dongarra, J. Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," ACM Transactions on Mathematical Software, Vol. 16, No. 1, pp. 1-17, March 1990.

[33] C. Bischof and C. Van Loan, "The WY Representation for Products of Householder Matrices," SIAM Journal of Scientific and Statistical Computing, Vol. 8, No. 1, pp. s2-s13, January 1987.

[34] R. Schreiber and C. Van Loan, "A Storage-Efficient WY Representation for Products of Householder Transformations," SIAM Journal of Scientific and Statistical Computing, Vol. 10, No. 1, pp. 53-57.