Eur. Phys. J. Special Topics **210**, 73–88 (2012)

DOI: 10.1140/epjst/e2012-01638-7

THE EUROPEAN PHYSICAL JOURNAL SPECIAL TOPICS

Regular Article

CPU vs. GPU - Performance comparison for the Gram-Schmidt algorithm

T. Brandes[1,a], A. Arnold[2], T. Soddemann[1], and D. Reith[1]

[1] Fraunhofer Institute SCAI, Schloss Birlinghoven, 53754 Sankt Augustin, Germany
[2] Institute for Computational Physics, University of Stuttgart, Pfaffenwaldring 27, 70569 Stuttgart, Germany

Received 30 April 2012 / Received in final form 25 June 2012
Published online 6 September 2012

Abstract. The Gram-Schmidt method is a classical method for determining QR decompositions, which is commonly used in many applications in computational physics, such as orthogonalization of quantum mechanical operators or Lyapunov stability analysis. In this paper, we discuss how well the Gram-Schmidt method performs on different hardware architectures, including both state-of-the-art GPUs and CPUs. We explain, in detail, how a smart interplay between hardware and software can be used to speed up those rather compute intensive applications as well as the benefits and disadvantages of several approaches. In addition, we compare some highly optimized standard routines of the BLAS libraries against our own optimized routines on both processor types. Particular attention was paid to the strong hierarchical memory of modern GPUs and CPUs, which requires cache-aware blocking techniques for optimal performance. Our investigations show that the performance strongly depends on the employed algorithm, compiler and a little less on the employed hardware. Remarkably, the performance of the NVIDIA CUDA BLAS routines improved significantly from CUDA 3.2 to CUDA 4.0. Still, BLAS routines tend to be slightly slower than manually optimized code on GPUs, while we were not able to outperform the BLAS routines on CPUs. Comparing optimized implementations on different hardware architectures, we find that a NVIDIA GeForce GTX580 GPU is about 50% faster than a corresponding Intel X5650 Westmere hexacore CPU. The self-written codes are included as supplementary material.

1 Introduction

A QR decomposition of a given matrix A is well-known as a decomposition in two factors Q and R, where Q is an orthonormal matrix and R is an upper triangular matrix. Since the orthonormal matrix can be interpreted as a simple coordinate transformation, R contains all relevant information on A, however, in a much simpler form.

[a] e-mail: Thomas.Brandes@scai.fraunhofer.de

Some uses of QR decompositions include solving linear systems of equations [1] and linear least square problems [2,3], or to determine eigenvalues and -vectors [4–6].

Besides the Gram-Schmidt method [7], there are two other widely used classical methods for determining a QR decomposition: Householder reflections [8] and Givens rotations [9]. The latter two methods successively eliminate matrix entries below the main diagonal of A, i.e. A is iteratively transformed into an upper triangular matrix R, while Q is given by the product of the reflections or rotations. In contrast to this, the Gram-Schmidt method transforms the matrix A column by column into an orthonormal matrix Q, while R is obtained from the coefficients of this transformation. The original Gram-Schmidt method itself is numerically unstable, since small rounding errors render the columns not exactly orthogonal. However, the commonly used modified version is stable [10], as well as the other two classical methods. While the Gram-Schmidt process is slower than Householder reflections or Givens rotations, it produce the first few orthogonal vectors immediately, while the other two methods need to complete before any columns of Q are orthogonal. This can be exploited for example in iterative Eigenvalue solvers based on Arnoldi iteration [11].

One common application of QR decompositions using the modified Gram-Schmidt process is the calculation of Lyapunov critical exponents of the time evolution of many particle systems. The determination of these exponents requires a QR decomposition of the variational equation associated with the underlying differential equation [12,13]. However, the matrix Q very quickly looses its orthogonality during the time evolution of the differential equation. Therefore, a frequent reorthogonalization of Q is necessary, which is typically done using the modified Gram-Schmidt method. This is much more time consuming than calculating the numerical solution of the differential equation, and usually limits the size of the systems that can be investigated to less than 1000 particles. Therefore, a speed-up of the QR decomposition is the key to allow investigations of larger and more realistic systems.

Such a speed-up can be obtained by employing hybrid hardware architectures, which have recently gained substantial attention, in particular compute architectures augmented by powerful graphics processors (GPUs). Such systems have, in short time, made their way into the top ten super computers of the world, and there are various QR decompositions available for GPUs [14,15]. Also the SIMD (single instruction multiple data) vector operations, given in all modern conventional processors (CPUs), can be exploited. However, all these accelerators require more hardware-aware programming, making it more difficult to harvest the peak performance of the present hardware. One needs a combination of state-of-the-art hardware and algorithms fitted to it. Also, since practically all current available hardware is multi-core, parallelization is no longer just a nice feature, but essential.

In this paper we first describe the modified Gram-Schmidt method and show how it can be parallelized specifically to take advantage of recent multi-core and GPU architectures. We compare various implementations using OpenMP on the CPU, a native GPU implementation using NVIDIA's Compute Unified Device Architecture (CUDA) [16], and versions using routines from the basic linear algebra subprograms library (BLAS) both on the CPU and GPU [17]. Importantly, we show how blocking techniques improve cache usage and how useful they are to reduce the memory traffic in the CPU and GPU versions.

2 The Gram-Schmidt method in a nutshell

We start by briefly recalling the modified Gram-Schmidt method [3]. Let A be a $m \times n$ matrix. We look for a QR decomposition $A = QR$, where Q is a $m \times n$ orthonormal matrix and R is an $n \times n$ upper triangular matrix. The modified Gram-Schmidt method computes such a QR decomposition by successively transforming all columns

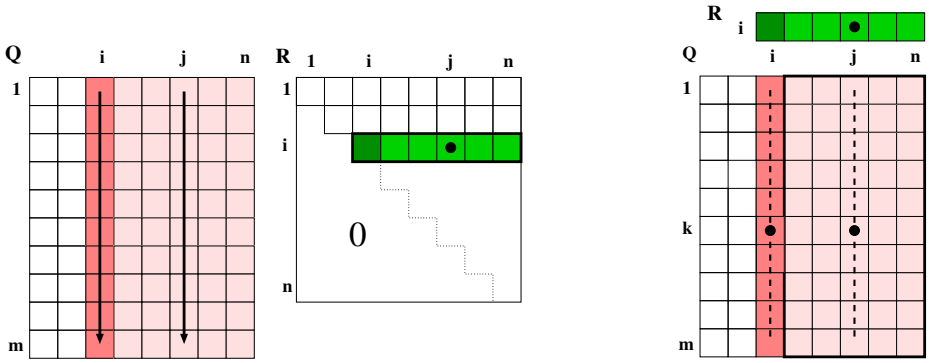

Fig. 1. Illustration of the computations in the Gram-Schmidt method, which gradually transform the matrix Q into an orthonormal matrix. The first $i-1$ columns of Q and $i-1$ rows of R are already computed, indicated by the non-filled elements. The i-th row of R is built by computing r_{ij} as the dot product of columns i and j of Q, depicted to the left. The square root of the first element r_{ii} is the norm of column i of Q and is used for its normalization and for scaling the dot products to get the factors for the projection. On the right is illustrated, how each remaining column j of Q is orthogonalized with respect to column i by subtracting its projection.

of A into orthonormal vectors. More precisely, the first column is normalized, and its projection onto all remaining columns is subtracted from each of them. By this, the first column is normal, and orthogonal to all other columns. This step we repeat for the second column, leaving the first untouched. Since the first and second column are orthonormal, and we subtracted their projections from all the $m-2$ other columns, they are both orthogonal to all the other columns. This procedure is repeated until all columns of A are orthonormal, i.e. until A has been transformed into an orthonormal matrix Q. R is obtained as a by-product from the projection coefficients. Figure 1 illustrates the computation of the i-th column of Q and the i-th row of R.

Let a_{kj} be the elements of A for $k = 1, \ldots, m$ and $j = 1, \ldots, n$, and correspondingly q_{kj} and r_{kj} the elements of Q and R, respectively. Initially, we choose $Q = A$, i. e. $q_{kj} = a_{kj}$ for all k, j. Then, the modified Gram-Schmidt method written in pseudo code looks as it is shown in listing 1, where the resulting Q is given by q_{kj} and R by r_{kj}. The total number of floating operations (additions and multiplications) spent during the orthogonalization and dot products steps is $2mn^2$, which to good precision is also the overall complexity of the algorithm, since the remaining n square roots and $nm + n(n-1)/2$ divisions require only marginal computation time. This allows us to define the performance in Giga floating point operations per second (GFlops), by which we denote the quantity $2mn^2/T$, where T is the execution time in seconds. Unlike T itself, this quantity is independent of the problem size and therefore easily allows to compare the performance for different problem sizes.

The implementation of the algorithm in C and Fortran is straightforward from the element-wise pseudo-code (listing 1). If the code is realized as a subroutine, the arrays Q and R will be passed as assumed-sized arrays in Fortran together with the values of the sizes m and n, allowing for two-dimensional indexing of the arrays. Unfortunately, C/C++ does not support assumed-sized arrays, which leads to the following corresponding code pieces with linearized two-dimensional indexes in C:

pseudo-notation:	$q_{i,j} = q_{i,j} - r_{k,j} q_{i,k}$
Fortran:	q(i, j) = q(i, j) − r(k, j) ∗ q(i, k)
C:	q[(j−1)∗m+i−1] −= r[(j−1)∗n+k−1] ∗ q[(k−1)∗m+i−1]

When analyzing the algorithm in vector-array notations instead (cf. listing 2), it is obvious that the algorithm can also be implemented by using Basic Linear Algebra

```
R = 0
for i = 1,...,n
    for j = i,...,n
        for k = 1,...,m
            r_ij = r_ij + q_ki q_kj
        end
    end
    r_ii = sqrt(r_ii)
    for k = 1,...,m
        q_ki = q_ki / r_ii
    end
    for j = i+1,...,n
        r_ij = r_ij / r_ii
    end
    for j = i+1,...,n
        for k = 1,...,m
            q_kj = q_kj - q_ki r_ij
        end
    end
end
```

Listing 1. Modified Gram-Schmidt method in element-wise notation.

```
R = 0
for i = 1,...,n
    ! dot products, matrix−vector mult
    ! R_{i,i:n} = Q^T_{1:m,i:n} × Q_{1:m,i}
    for j = i,...,n
        r_ij = Q_{1:m,i} · Q_{1:m,j}
    end
    ! normalize column i of Q
    s = sqrt(r_ii) // norm of i−th column of Q
    Q_{1:m,i} = Q_{1:m,i} / s

    ! compute projection factors
    R_{i,i:n} = R_{i,i:n} / s

    ! orthogonalization, rank−1 update
    ! Q_{1:m,i+1:n} − = Q_{1:m,i} × R^T_{i,i+1:n}
    for j = i+1,...,n
        Q_{1:m,j} = Q_{1:m,j} − r_ij Q_{1:m,i}
    end
end
```

Listing 2. Modified Gram-Schmidt method in vector notation.

```
subroutine QR(q, r, m, n)
  integer i, m, n
  real q(m,n), r(m,n), S
  do i = 1, n
    ! R_{i,i:n} = 1.0 * Q^T_{1:m,i:n} × Q_{1:m,i} + 0.0 * R_{i,i:n}
    call sgemv(trans = 'T', m = m, n = n-i+1, alpha = 1.0, A = q(1,i),
               lda = m, x = q(1,i), incx = 1, beta = 0.0, y = r(i,i), incy = n)
    S = 1.0 / sqrt(r(i,i))
    call sscal(n = m, alpha = S, x = q(1,i), incx = 1)
    call sscal(n = n-k+1, alpha = S, x = r(i,i), incx = n)
    ! Q_{1:m,i+1:n} = −1.0 * Q_{1:m,i} * R^T_{i,i+1:n} + Q_{1:m,i+1:n}
    call sger(m = m, n = n-i, alpha = -1.0, x = q(1,i), incx = 1,
              y = r(i,i+1), incy = n, A = q(1,i+1), lda = m)
  end do
end subroutine QR
```

Listing 3. Modified Gram-Schmidt method using BLAS.

Subprograms (BLAS) routines, a well known API standard for publishing libraries to perform basic linear algebra routines [17]. Highly tuned library implementations are available on nearly all platforms, like the Intel Math Kernel Library (Intel MKL) for Windows, Linux, and MacOS, which is especially optimized for Intel processors. The loop over the dot products is equivalent to a matrix-vector multiplication for which the BLAS-2 routine GEMV (general matrix-vector multiplication) is available, that also supports the operation for a transposed matrix as required here. The orthogonalization is a rank-1 matrix update, which is realized by the BLAS-2 routine GER. The normalization and the computation of the projection factors are just vector scalings, supported by the BLAS-1 routine SCAL. Listing 3 shows the implementation of the algorithm using BLAS routines. Arrays and vector arguments are passed by a pointer

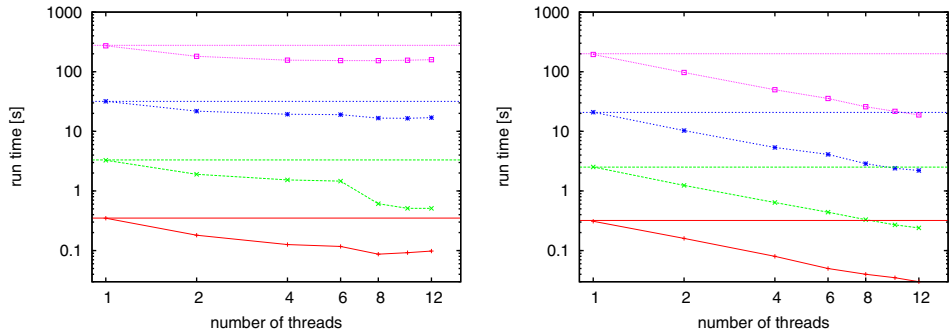

Fig. 2. Execution time in seconds of an OpenMP parallel CPU version, depending on the number of threads employed. Results are, from top to bottom, for matrix sizes 8k × 8k, 4k × 4k, 2k × 2k and 1k × 1k (k denotes 1024). Straight lines give the result from the serial version not using OpenMP, which compare well to the one thread OpenMP results, showing that the OpenMP overhead is negligible. The left graph is for the simple, non-blocked version, the right graph for the blocked version, which shows a nearly ideal scaling with the number of threads.

to the first element and by the stride. The BLAS-routines can also be called from a C code, but for matrices the column-major storage order must be observed.

3 CPU parallelization and blocking

OpenMP parallelization: The OpenMP parallelization of the code (see listing 1) is quite straightforward. The outermost loop is not parallelizable, as it has data dependencies for the columns: each column is updated by values computed in the previous iteration. But all loops on the next level are independent, i.e. each of the corresponding array operations can be carried out fully in parallel. Therefore these loops just need to be annotated with the *parallel for* pragma (C) or *parallel do* directive (Fortran) provided by OpenMP.

We measured the performance of the OpenMP implementation on one CPU node with two Intel Xeon X5650 CPUs (Westmere architecture, 2.67 GHz, 6 cores, 12 MB L3 cache). As explained below, we have used the Fortran version compiled with the Intel compiler suite (12.0.2). Up to 6 threads were always bound to one single CPU to avoid any memory placement issues and thread migration. The left graph of Fig. 2 shows the results. The scaling of the shared memory parallelization is very poor. The explanation is that in each iteration both computations, the dot products and the orthogonalization, traverse all remaining columns of Q. The amount of data stored in these columns is so large that it does not fit in the cache and has to be reloaded for each iteration. Hence, the memory bandwidth becomes the limiting factor.

In order to reduce the required memory bandwidth we have implemented a blocked version of the code, cf. fig. 3. The loop range of the outer block is divided into blocks where each block has the size b. This way, b columns of the result matrix Q are computed before any other updates of the remaining matrix are done. As the old columns are still needed for the dot products, they must be saved for each block. In the update part of the method, which now clearly dominates the computation time, each column can be processed independently by using the old b columns for the dot products and the new b columns for the projection. If the j-th column is kept in a local cache and the b old and new columns in a global cache, memory traffic is reduced at least by a factor b. The blocked code, in which only the order of computations is

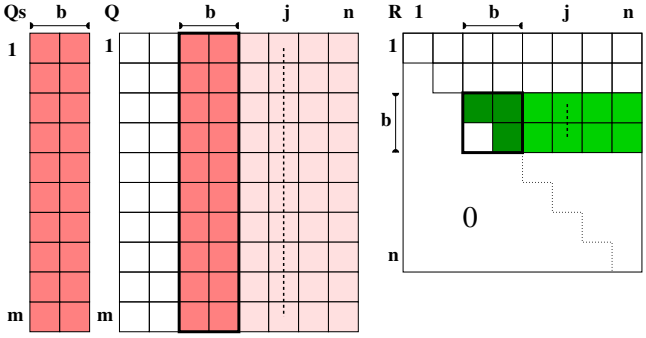

Fig. 3. Illustration of the blocking technique. The QR factorization is applied to b columns of Q resulting in b orthogonal columns and a submatrix of R with size $b \times b$ (framed parts). The remaining projection factors in the b rows of R are computed afterwards together with the orthogonalization of the remaining columns of Q. Both can be done for each column j independently and therefore in parallel. The original b columns of Q are still needed for the dot products while the b new columns are used for the orthogonalization. Blocking gives a better cache usage as explained in the text.

changed, is as follows:

for $i1 = 1, b+1, 2*b+1, \ldots, n$
  $i2 = \min(i1+b-1, n)$
  $Q^s(1:m, 1:b) = Q(1:m, i1:i2)$
  call QR(Q(1:m,i1:i2), R(i1:i2,i1:i2))
  for $j = i1+b, \ldots, n$ // parallel loop over remaining columns
    for $i = i1, \ldots, i2$
      $r_{ij} = Q^s_{1:m,i-i1+1} \cdot \mathbf{Q_{1:m,j}}$
      $r_{ij} = r_{ij} \;/\; r_{ii}$
      $\mathbf{Q_{1:m,j}} \;-\!= r_{ij} * Q_{1:m,i}$
    end
  end
end

Although the number of floating-point operations has not changed with respect to the nonblocked version, the performance of this code is drastically different on multi-core systems. The achieved speed ups of the blocked version are shown in the right graph of fig. 2. We now see ideal speed ups up to the maximum number of 12 cores on a node with two Xeon X5650 CPUs for all problem sizes, showing that the bottle neck was indeed due to insufficient cache exploitation. The block size b is a tuning parameter that requires optimization, see fig. 4. Wrong choices of the block size can reduce the performance by up to a factor of four. In any case, the blocking factor should guarantee that the b buffered old and new columns fit in the global L3 cache. As there is no obvious relation between problem size and optimal block size, we realized the choice of b by a look-up table, which is hardware-dependent.

As only one-dimensional blocking is applied, larger problem sizes can not be as fast as smaller ones. Generally speaking, two-dimensional blocking is more suited for efficient parallelization due to better cache usage, especially for larger problems. Very often one can achieve nearly peak performance, which is demonstrated by the BLAS-3 routines that are based on this technique. However, two-dimensional blocking is not possible for the modified Gram-Schmidt algorithm. That is due to the fact that the orthogonalization for each column of Q forms a recurrence over the full column, i.e.

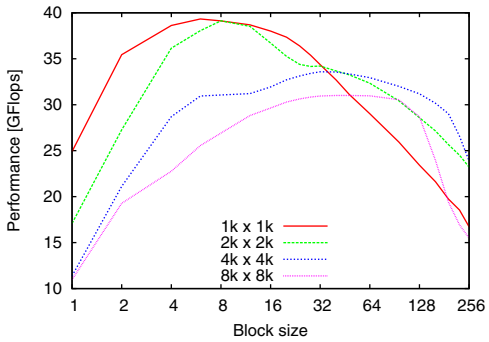

Fig. 4. Achieved performance in GFlops for different problem and block sizes b on a single Xeon X5650 (6 cores). The best performance for smaller problems is higher than for larger problems as one updated column can be kept completely in smaller and faster caches.

the update of one column j in iteration i

$$\mathbf{Q^{i}_{1:m,j}} = Q^{i-1}_{1:m,j} - [\mathbf{Q^{i-1}_{1:m,j}} \cdot Q^{i-1}_{1:m,i}/r_{i,i}] \times Q^{i}_{1:m,i} \tag{1}$$

needs all values of the same column from the previous iteration, $Q^{i-1}_{1:m,j}$, used in the dot product to compute the scaling factor r_{ij}. Consequently, corresponding loops over the elements of one column cannot be used for blocking. As two-dimensional blocking is not possible, BLAS-3 routines can not be exploited for the modified Gram-Schmidt method either.

Compiler dependence: In this section we would also like to point out how important the right choice of the programing language and the compiler is. We measured Fortran and C versions of the code with different versions of GNU and Intel compilers and compared it against a version using BLAS routines of the Intel MKL library for the inner most loops, as described in the previous section. Figure 5 shows the execution times in seconds for different problem sizes, measured on one core of a Intel Xeon X5650 CPU using the toolkits as described in table 3. The Intel Fortran Compiler and the MKL significantly outperform other compilers by a factor of two to three, depending on the matrix size. Even though the GNU C compiler collection has dramatically improved recently, it is less performant than the Intel compilers, both in Fortran and C.

For both compiler suites, Fortran versions are faster than the C versions. This is due to the linearization of indexes in C, which makes it harder for the compiler to optimize the code. We verified this by using fixed-size arrays in C, where the sizes of the arrays are given at compile time. With such arrays the Intel C compiler generates nearly as efficient code as the Intel Fortran compiler, the same is true for the GNU compilers. The use of fixed-size arrays is however very inflexible and often impossible, for example in a library. Therefore, for applications with multi-dimensional arrays Fortran remains the first choice when performance is important.

Table 3 compares the execution times of the parallel unblocked and blocked version using up to 12 cores. The parallel unblocked version only scales up to the memory bandwidth limit and all versions have nearly the same performance running with 6 or 12 cores. The blocked version scales very well, for all compiler versions and also for two CPUs with 12 cores. The Intel Fortran compiler is the only compiler where we see a large improvement for the serial version due to the blocking (40%) while all other compilers show only small improvements (up to 8%). For the parallel blocked implementation, the Intel Fortran version is about 2.5 faster than the Intel C version,

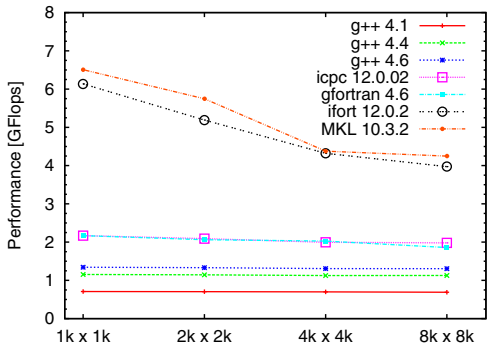

Fig. 5. Performance in Gigaflops of a serial CPU version compiled using various compilers on a Intel Xeon X5650 @ 2.67 GHz CPU, depending on the matrix size. MKL denotes results using the Intel MKL library for the central loops.

Table 1. Execution times (in seconds) of the unblocked and blocked version for problem size 8k × 8k. Results are given for the serial and for the parallel version running on one (6 threads) or two CPUs (12 threads). The MKL is not parallelized, therefore results are reported for the serial code.

compiler	g++	g++	icpc	gfortran	ifort	MKL
version	4.1	4.6	12.0.02	4.6	12.0.2	
flags	-O5	-O5	-fast	-O5	-fast	
unblocked, serial	1057.41	878.15	505.58	538.47	276.81	258.78
unblocked, 6 threads	192.81	175.06	156.05	162.00	153.84	
unblocked, 12 threads	158.62	190.33	157.54	200.02	159.01	
blocked, serial	1025.72	841.56	507.16	498.20	195.33	
blocked, 6 threads	178.84	145.15	88.05	110.11	35.55	
blocked, 12 threads	95.33	76.28	45.35	56.58	18.86	

about 3 times faster than the GNU Fortran version, and about 4 times faster than the GNU C version. On one core, the Intel MKL performs only slightly better than the Intel Fortran version, which indicates that the Fortran compiler either uses the MKL routines, or generates similar code.

4 Native GPU implementation

NVIDIA's CUDA framework [16] allows to use versions of the C or C++ programming languages extended by the domain specific language C for CUDA for programming the highly parallel hardware architecture of GPUs. In a nutshell, a GPU consists of several processors which can only communicate through the GPU's global memory. Each processor executes threads concurrently (at present up to 1024), which are bundled in thread blocks that can synchronize and communicate through a small amount of dedicated shared memory (at present 48 kilobytes per processor). Each computational task, called a kernel, is executed concurrently in a SIMD fashion by the thread blocks. Several thread blocks are combined to form the computation grid. Thread blocks in this computation grid may be executed independently and thus in parallel. They may even be assigned automatically to different processors by the compute hardware.

Access to the global memory is almost an order of magnitude slower than access to the shared memory, although it is still almost an order of magnitude faster than the

```
__global__
void Orthogonalization(Q, R, m, n, i)
{

    tx = threadIx.x;
    ty = threadIx.y;
    j  = blockId * NY + ty + i + 1;

    for (k=tx+1; k <= m; k+=NX) {


        q_kj = q_kj - r_ij q_ki;
    }
}
```

Listing 4. direct access orthogonalization.

```
...
...
{
    __shared__ ri[NY], qi[NX];
    ...
    ...
    ...
    if (tx == 0) ri[ty] = r_ij;
    for (k=tx+1; k <= m; k+=NX) {
        if (ty==0) qi[tx] = q_ki;
        __syncthreads();
        q_kj = q_kj - ri[ty] * qi[tx];
    }
}
```

Listing 5. shared memory orthogonalization.

main memory of conventional CPU architectures. However, this bandwidth is mainly realized due to a very wide memory data bus, which reaches its peak throughput only if the memory access patterns match the memory layout. On early GPUs of compute model 1.0, the base address of the memory accessed by a block had to be aligned to 128 bytes, and threads where required to read successive words from the memory for optimal bandwidth utilization. This restriction is much less stringent in recent hardware, where caches can store unused values read from the memory. Nevertheless, these values should be used, therefore an optimized memory layout of the data structures is advisable on GPUs. GPUs with compute model 1.0 offer a 16 kB read cache per processor, which was only usable through special read instructions, namely texture fetches. In contrast, each processor of the current Fermi architecture has a 16 kB L1 cache, that can be increased to 48 kB if less then 16 kB of shared memory are used, and 768 kB L2 cache for all global memory accesses.

Even slower than the global memory, and in fact even slower than the memory of a conventional CPU, is the transfer of data from the CPU to the GPU via the bottleneck of the PCIe bus. However, for the implementation of the Gram-Schmidt algorithm, only the input matrix needs to be copied from the host to the device, and the result matrices Q and R have to be retrieved from the device. This memory transfer is of order $O(mn)$ and the computation time costs are negligible compared to the time needed for the QR factorization.

The outer loop of the algorithm is not parallelizable and, therefore, is implemented as serial loop in the GPU code, too. Also the four different computations in the inner loop are implemented as separate kernels due to the global synchronization which is needed between the computations. The single square-root calculation to obtain the norm of q_i is joined with the dot product kernel, the other kernels implement the two scalings and the orthogonalization, as described in listing 2.

Out of these kernels, only the dot products and orthogonalization contribute significantly to the overall computation time, since they are the two kernels that scale as $\mathcal{O}(nm)$. In detail, our implementations are as follows. Note, that since most operations are columnwise, we store that matrix in column-major order.

Orthogonalization: Each thread block is responsible for the update of NY columns, so that all thread blocks can be executed independently, and consists of (NX, NY) threads. Thread (tx, ty) updates the values (i*NX+tx+1,bx+ty), $i = 0, 1, \ldots m/\mathrm{MX}$ of Q, where bx denotes the first column assigned to the thread block. The code is

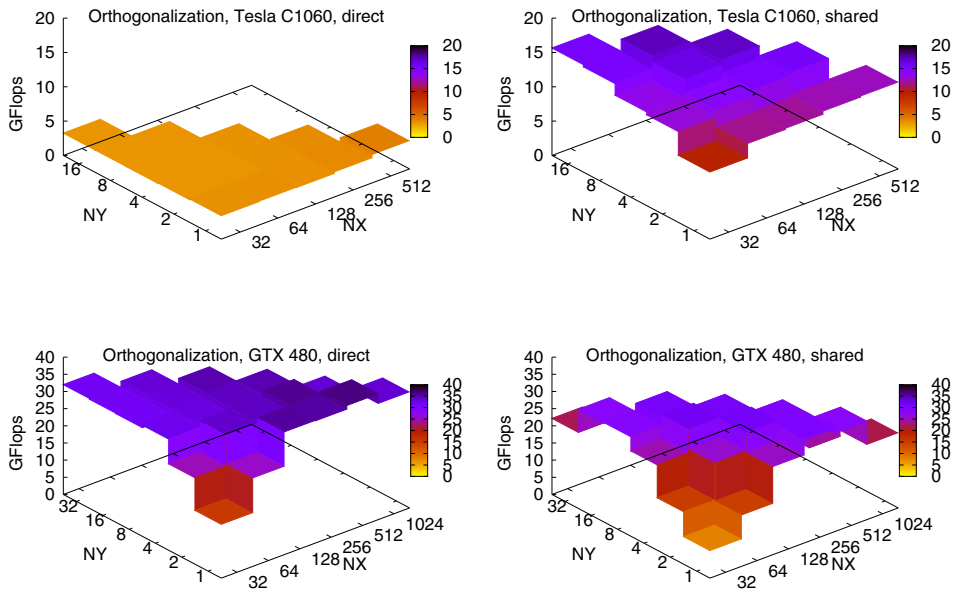

Fig. 6. Achieved performance of the orthogonalization kernel for different thread block sizes on a Tesla C1060 (top) and GeForce GTX480 (bottom). Left graphs show results for the direct access code, right graphs using shared memory. The best performance of the C1060 with direct access is only 4.5 GFlops (thread block shape 512×1), which improves to 17.3 GFlops (64×8) using shared memory. The GTX480 reaches 36.1 GFlops (512×1) using direct access, and 31.9 GFlops (64×4) using shared memory.

shown in listing 4. Note that we use mathematical notations q_{kj} to denote the linearized access to the two-dimensional array Q as described in Sect. 2.

Due to stride one the global memory access by all threads of a half-warp is coalesced, provided of course that NX ≥ 16. Therefore we can expect maximal bandwidth to the global memory for the write access even on older, non-caching hardware. For the values of the right hand side this is not true as threads with the same row or column index access the same value, although of course in later steps, each core has to access all adjacent values. This can be remedied by manual caching, that is, loading of the values into the shared memory. They are read by only one thread into shared memory, which can be read by all other threads after synchronization (see Listing 5).

The choice of NX and NY for the shape of the thread block can be used for performance tuning. There is an upper limit of 512 (for Tesla C1060) or 1024 (Geforce GTX480) threads due to hardware limitations. Smaller block sizes allow to run more thread blocks per multiprocessor, but as the number of thread blocks per multiprocessor is also limited (8 for all GPUs), too small block sizes result in a poor occupancy. Figure 6 shows the results on the C1060 and the GTX480 for different thread block sizes. For our tuning tests, we always used a rather large matrix size of $8k \times 8k$ in order to minimize the influence of minor fluctuations in the computation time.

On the C1060, the performance of the shared memory version is about a factor of 4 better, which is clearly due to the lack of efficient caches. The best performance is 17.3 GFlops for a thread block size of 64×8. Here we can also demonstrate that it is not sufficient to assign only one column to a thread block (NY $= 1$). With NY $= 1$, only 12.7 GFlops can be achieved (thread block size 512×1). As all columns need

```
__global__
void Dotproducts(Q, R, m, n, i)
{
    __shared__  RS[NY][NX];

    tx = threadIx.x;
    ty = threadIx.x;
    j  = blockId * NY + ty + i;
    sum = 0;
    for (k=tx+1; k<=m; k+=NX) {



        sum += q_ki q_kj;
    }
    // reduction: r_ij += sum;
    RS[ty][tx] = sum;
    NT = NX;
    while (NT > 1) {
        __syncthreads();
        NT = NT / 2;
        if (tx < NT)
            RS[ty][tx] += RS[ty][tx+NT];

    }
    if (tx==0) r_i,j = RS[ty][0];
}
```

Listing 6. direct access dot products.

```
...
...
{
    ...
    __shared__ qi[NX];
    ...
    ...
    ...
    ...
    for (...) {
        if (ty==0) qi[tx] = q_ki;
        __syncthreads();
        sum += qi[tx] * q_kj
    }
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
}
```

Listing 7. shared memory dot products.

the values of column i, the number of reads is reduced by this factor NY. Generally speaking, the term NX + NY should be minimized to reduce the memory traffic for both vectors needed in the rank-1 update.

On the GTX480, the results are completely different. Using the shared memory decreases the performance slightly as the shared memory code simply requires more instructions, without doing better caching than the hardware does. The hardware actually caches better, because the cached values might even be reused for different thread grid blocks running on the same multiprocessor. Since there is no overhead when dealing with a small number of columns NY, the simple case of a block of 512×1 is optimal, reaching 36.1 GFlops. This seems far from the peak performance of the device, but the Gram-Schmidt method is memory-bound. For each update of q_{kj}, that is, two floating point operations, four bytes have to be read and written. 36.1 GFlops therefore correspond to a memory throughput of 144 GB/s which is close to the memory bandwidth of the GTX480.

Dot products: Each thread grid block is responsible for calculating NY dot products. Every dot product is processed by NX threads, so that multiple threads contribute to each value r_{ij}. Such a reduction operation can exploit the atomic updates present on device of compute capability 2 and above, however, a hierarchical reduction in shared memory is more efficient [18]. Listing 6 shows our implementation, which performs the summation in $\log_2(NX)$ steps, reducing the number of working threads by a factor of 2 each step. Listing 7 shows a version that also uses shared memory to buffer the

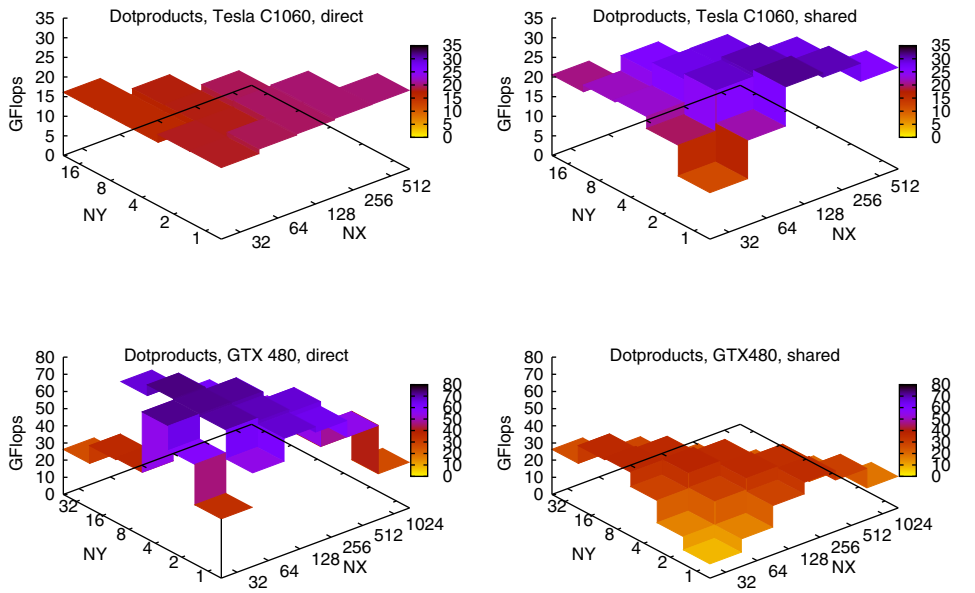

Fig. 7. Achieved performance for the dot product kernel on Tesla C1060 (top) and GeForce GTX480 (bottom) with different thread block shapes. Left graphs show results for the direct access code, right graphs using shared memory. Without shared memory, the C1060 reaches a maximal performance of 20.1 GFlops (shapes with $NX \geq 256$), which increases to 31.6 GFlops (128×1). The GTX480 reaches 73.3 GFlops (64×8) with direct access, which reduces to 38.1 GFlops (64×4) when buffering q_{ki}.

value q_{ki}, similar as described for the orthogonalization kernel. Again, the thread block sizes NX and NY can be used for performance tuning. To guarantee coalesced memory access, NX should be at least half warp size, i. e. NX$\geq$ 16.

Figure 7 shows the performance on the Tesla C1060 and GeForce GTX480 for the kernel computing the dot products for different thread block sizes. The use of shared memory gives a performance gain on the C1060 (factor 1.5), but not as high as it was for the orthogonalization (factor 4). For the GTX480, buffering the value q_{ji} in shared memory is again not helpful. Here, the effect is more pronounced, the performance decreases by a factor of nearly 2.

Comparing the dot product and orthogonalization kernels for each device, one can see that the dot product kernel (31.6 GFlops on the C1060, 73.3 GFlops on the GTX480) has nearly twice the performance than the orthogonalization kernel (17.3 GFlops on the C1060, 39.3 on the GTX480). Formally, both kernels have the same number of floating point and read and write operations, namely two read accesses and one update for two floating point operations. In both cases, q_{ki} is heavily reused and therefore does not require much memory bandwidth. However, the summation of the dot product kernel happens mostly in registers and shared memory and is therefore negligible, while the rank-1 update of the orthogonalization happens in global memory. Therefore, the memory traffic of the orthogonalization kernel is twice as high, which explains the doubled performance and again shows that the Gram-Schmidt method is memory-bound.

The blocking technique applied for the CPU version could also be applied for the GPU version. We implemented an additional update kernel where one thread grid block updates one remaining column of Q with the old and new b columns. For larger problems, we could not see any benefits as the local cache (maximal 48 kB on one

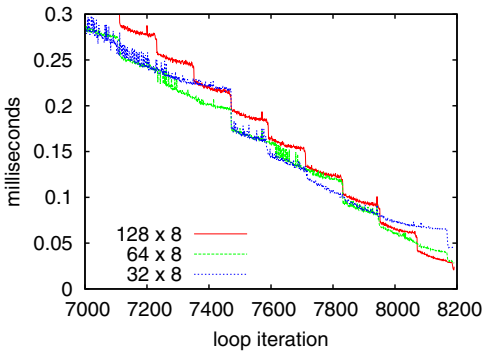

Fig. 8. Execution times of GEMV kernel as a function of the iteration using different thread block shapes. Since the remaining matrix becomes smaller, the computation decreases with iteration number. Steps are caused by the decreasing number of employed thread blocks.

multiprocessor) does not even fit to hold one column or the need of shared memory results in poor occupancy. For matrices with smaller column size m, there was a performance gain of about 50%, but obviously the cache did not fit to keep the old and new b columns used by all thread blocks.

As said before, we were always optimizing the total execution time of the kernel at a large problem size of $8k \times 8k$. However, the tuning might be different if smaller problem sizes are considered, and the optimal thread topology might be different for different iterations of the outer loop. This is indeed the case, as Fig. 8 shows, where we plot the execution times of single kernel calls to the orthogonalization kernel during the last iterations for a $8k \times 8k$ matrix. The results were measured on the GTX480 where the thread block size 64×8 gives the best total performance if only one topology is used. During the last iterations, when the problem size becomes smaller, it can be seen that other topologies like 32×8 or 128×8 give better performance. However, the differences are marginal, so that it is not worth to optimize for each matrix size that, given the considerable overhead this would mean.

5 CUBLAS

CUBLAS (Compute Unified Basic Linear Algebra Subprograms) is an implementation of BLAS for GPUs based on NVIDIA CUDA [19]. The library is self-contained at the API level, that is, no direct interaction with the CUDA driver is necessary. Since CUBLAS uses column-major storage and 1-based indexing like Fortran and BLAS, porting the BLAS version (listing 3) from the CPU to the GPU mainly requires to replace the BLAS calls with CUBLAS calls and using the CUBLAS functions for memory allocation and transfer between CPU and GPU.

Figure 9 compares the performance of different versions of the CUBLAS library with our hand-written code. The native CUDA implementation outperforms the CUBLAS implementation in all versions, although the latest release, 4.1, allows to achieve about 90% of the performance of our native implementation on the Fermi architecture. However, on the older Tesla architecture, it reaches only 55%. This is caused by the CUBLAS routine GER used for the orthogonalization, which has nearly the same performance in all versions of CUBLAS (3.2 from Nov 2010, 4.0 from May 2011, 4.1 from Dec 2011). While it is nearly the same as our CUDA orthogonalization kernel with direct access on the Fermi architecture, its performance on the Tesla is only about half of the performance of our kernel with shared memory buffering.

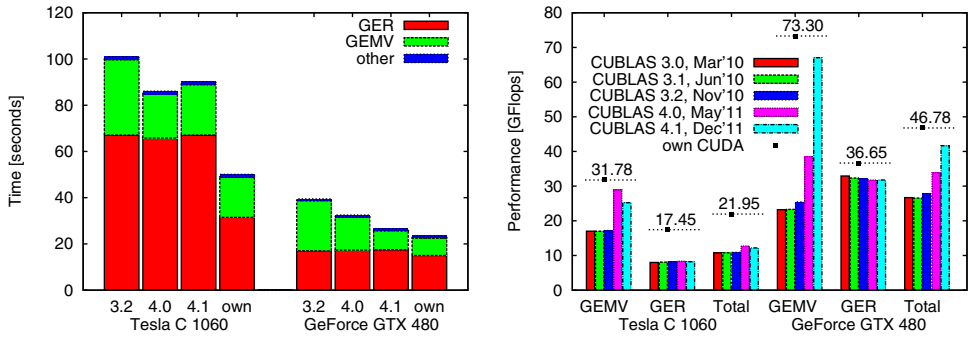

Fig. 9. Comparison of our CUDA version with the CUBLAS version using different CUBLAS versions. Left graph shows cumulative execution time, right graph the performance.

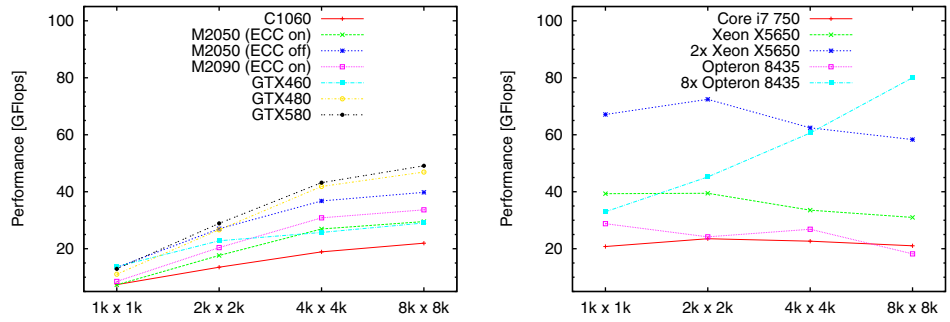

Fig. 10. Performance of our Gram-Schmidt implementation using different processors (left: GPUs, right: CPUs), cf. table 2.

The CUBLAS routine GEMV for the dot products has improved in the newer releases of the CUBLAS library. In other applications we could see that GEMV for normal matrices was more efficient already in earlier releases of CUBLAS, so GEMV for transposed matrices was probably not in the focus. The performance of GEMV in release 4.1 is now comparable to both our CUDA dot product kernels, the one with direct access for the Fermi architecture and the one with shared memory for Tesla.

6 Comparison of CPU and GPU

In this section we compare the performance of the Gram-Schmidt method on different devices, GPUs and multi-core CPUs. For the GPU we used the CUDA implementation as described in section 4, using shared memory buffering on older GPUs and direct memory access on the Fermi architecture. The dimensions of the thread grid blocks were set individually to maximize the performance. On CPUs we used the blocked OpenMP version implemented in Fortran and the Intel Fortran compiler (12.0.2) with a tuned blocking factor. The main hardware features of the used GPUs and CPUs are listed in table 2.

Figure 10 shows the performance of our code on CPUs and GPUs. The fastest GPU (GeForce GTX580) outperforms a single multi-core CPU (Xeon X5650) by about 50%, but is already slower than two multi-core CPUs on one node. Surprisingly, the 8×6 core AMD Opteron system does not perform significantly better than the 2 Xeon processors, and is faster only for the largest matrix size. In contrary to the CPUs, the performance of the GPU version is limited by the memory bandwidth. This can

Table 2. Used hardware configurations of NVIDIA GPUs (top) and CPUs (bottom).

Name	Compute Capability	MP	Cores / MP	CUDA Cores	Proc Clock MHz	Mem Clock MHz	Mem Width bit	Global Mem GB
C1060	1.3	30	8	240	600	1600	512	4
M2090	2.0	16	32	512	650	1850	384	6
M2050	2.0	14	32	448	573	1546	384	6
GTX460	2.1	7	48	336	675	1800	256	1
GTX480	2.0	15	32	480	700	1848	384	1.5
GTX580	2.0	16	32	512	772	2004	384	3

Name	CPU MHz	Cores	L2-Cache kB	L3-Cache MB
Intel Xeon X5650	2666	6	6 x 256	12
Intel i5 750	2660	4	4 x 256	8
AMD Opteron 8435	2660	6	6 x 512	6

be seen for example by the fact that switching off ECC on the M2050 improves the performance by 30%. While we achieve approximately half of the Linpack performance on a CPU, we get only one-tenth of it on the GPUs.

7 Summary and conclusions

We implemented and optimized the modified Gram-Schmidt algorithm for multi-core CPUs and for GPUs. Even though the parallelization strategy of the Gram-Schmidt algorithm is easy and rather straightforward for both kinds of devices, the performance of thee simple solutions remained far behind the expectations.

For good benchmark results on a multi-core CPU it is absolutely mandatory to apply blocking techniques as memory bandwidth would be otherwise a severe limiting factor. Also the right choice of programming language and compiler can result in a performance speed-up factor of up to 5. As C/C++ does not support assumed-sized arrays like Fortran, this language is not a good choice for the implementation of matrix algorithms like the Gram-Schmidt method; fixed sized arrays with multi-dimensional indexes can solve the problem, but are too inflexible for practical uses. Additionally, highly optimizing compilers like the Intel compilers clearly outperform free compilers like the GNU compilers.

For the implementation on a GPU, coalesced memory access and the use of shared memory are the most important issues for performance tuning. With the new generation of GPUs, the use of shared memory is much less critical for a good performance. In cases where multiple threads read the same value, the use of the shared memory can reduce the performance due to the required synchronization, in contrast to pre-Fermi hardware, where shared memory was a key to performance. Writing code for the current Fermi architecture is therefore easier, but codes optimized for pre-Fermi architectures might no longer be optimal for the Fermi architecture. Therefore, software development for GPUs will focus on the new architectures as it is obviously already the case for the CUBLAS library.

In both cases, GPUs and CPUs, gaining maximal performance requires careful tuning to fit the cache and memory hierarchies. On CPUs, the key is efficient exploitation of the large caches by appropriate blocking. On GPUs, one has to evaluate different memory access patterns due to different strides and compute capabilities.

Good heuristics for choosing optimal access patterns are very important for performance optimization. BLAS routines are a reasonable comprise as they provide good, although not optimal performance both on CPUs and GPUs and a simple implementation. On CPUs this is especially true for C programs, which the investigated compiler families optimize much less than Fortran codes. On GPUs, the BLAS-2 routines have improved and are new comparable to well optimized hand-tuned code, although only on recent hardware.

Comparing the modified Gram-Schmidt method implemented on CPUs and GPUs, the effort to write performant code is comparable. However, completely different approaches are necessary – blocking of the outer loop on the CPU and distribution in thread blocks on the GPU. The fastest GPU, a GeForce GTX 580, is only about 50% faster than the fastest CPU, a Xeon X5650 hexacore, and cannot compete with a standard dual processor node, which is much less than what one would expect from comparing with Linpack, for example. The reason is that the Gram-Schmidt method only allows blocking along one dimension of the matrix, which requires large caches to be able to store several columns. These are at present only available on CPUs, so that a GPU implementation is bounded by the memory bandwidth. Insofar the modified Gram-Schmidt is an example of a compute-intensive method that nevertheless cannot benefit much from present GPUs.

We thank Dustin Feld for providing performance results with the latest GNU compilers and Jiri Kraus for many fruitful discussions about efficient usage of NVIDIA GPUs.

References

1. J. Stoer, R. Bulirsch, W. Gautschi, C. Witzgall, *Introduction to Numerical Analysis* (Springer, 2002)
2. Å. Björck, BIT Numer. Math. **7**, 1 (1967)
3. G. Golub, C. Van Loan, *Matrix Computations* (John Hopkins University Press, 1996)
4. J. Francis, Comput. J. **4**, 265 (1961)
5. J. Francis, Comput. J. **4**, 332 (1962)
6. V. Kublanovskaya, Comput. Math. Phys. **3**, 637 (1961)
7. J. Gram, J. Math **94**, 45 (1883)
8. A. Householder, J. ACM (JACM) **5**, 342 (1958)
9. W. Givens, National Bureau Stand. Appl. Math. Ser. **29**, 117 (1953)
10. E. Schmidt, Math. Annal. **63**, 433 (1907)
11. W. Arnoldi, Quart. Appl. Math. **9**, 17 (1951)
12. H. von Bremen, F. Udwadia, W. Proskurowski, Physica D: Nonlinear Phenomena **101**, 1 (1997)
13. F. Christiansen, H. Rugh, Nonlinearity **10**, 1063 (1997)
14. *CULA Programmer's Guide Release 13 (CUDA 4.0)*, EM Photonic, Inc., Newark, DE (2011)
15. A. Kerr, D. Campbell, M. Richards, *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (ACM, 2009), p. 71
16. *CUDA Programming Guide*, NVIDIA, Santa Clara, CA (2011)
17. J. Dongarra, I. Duff, D. Sorenson, D. Sorensen, H. van der Vorst, *Numerical Linear Algebra on High Performance Computers (Software, Environments, Tools)* (SIAM, 1999)
18. M. Harris, *Optimizing Parallel Reduction in CUDA*, NVIDIA white paper, Santa Clara, CA (2008)
19. *CUDA Toolkit 4.0, CUBLAS Library*, NVIDIA, Santa Clara, CA (2011)