Chapter 4
Efficient Batch LU and QR Decomposition on GPU

William J. Brouwer and Pierre-Yves Taunay

4.1 Batch LU Decomposition

While comparatively expensive, direct solvers based around matrix decomposition are used in various applications, for reasons of numerical stability, over iterative solvers. The implementation presented shortly was originally devised for the solution of many decoupled systems simultaneously [4], for what amounts to a domain decomposition approach [6]. The LU decomposition also provides a viable method for the calculation of the matrix determinant; after execution of an in-place implementation, the determinant is available from the product of the diagonal elements. This is particularly useful in condensed matter physics, specifically in studies of the fractional quantum Hall effect based on construction of the Pfaffian wave function, which requires $O(N!)$ determinant evaluations [9, 10].

4.1.1 Theory

The decomposition of matrix A into lower L (elements α_{ij}) and upper U (elements β_{ij}) matrix,

$$\mathbf{LU} = \mathbf{A}, \tag{4.1}$$

has the advantage of permitting the solution of linear systems in two steps, comprised of forward and backward substitution procedures, for multiple right hand sides in $Ax = y$. Crout's approach to LU decomposition solves the set of equations implicit to Eq. (4.1); these are:

W.J. Brouwer (✉) • P.-Y. Taunay
The Pennsylvania State University, University Park, PA, USA
e-mail: wjb19@psu.edu; py.taunay@psu.edu

V. Kindratenko (ed.), *Numerical Computations with GPUs*,
DOI 10.1007/978-3-319-06548-9_4,

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}, \tag{4.2}$$

and

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right). \tag{4.3}$$

Numerical stability relies on suitable choice of pivot, or dividing element in the solution for α_{ij}. Pivoting may be partial (a row interchange) or full (both row and column); the former is implemented in this chapter. Following the approach detailed in Numerical Recipes [5], the choice of the best pivot is made only after both Eqs. (4.2) and (4.3) are solved for a given column, and thereafter the row swap and a scaling performed. Recording the row permutations in a separate vector is required for use with the solution of linear equations, in order that the right hand side vector be subsequently rearranged to suit. Equations (4.2) and (4.3) give rise to $N^2 + N$ equations, whose overdetermined nature permits the setting of N elements arbitrarily. A popular choice is to set the diagonal elements of α to one, followed in this chapter. Crout's approach to LU decomposition is summarized in Algorithm 1.

4.1.2 GPU Implementation

With the foreknowledge that the decomposition will be applied in batch, the mapping of computational thread to matrix is a seemingly reasonable strategy for a GPU implementation. However, on the device this virtually eliminates the possibility of coalesced loads from global memory, and thread cooperation via shared memory, key requirements for good performance. At the other extreme, mapping thread to matrix element would introduce significant overhead in the form of synchronization, owing to dependencies between the loops described in Algorithm 1. In a compromise between the two extremes, $O(N)$ threads were assigned to the operations for each matrix, and individual CUDA thread blocks assigned one or more matrices to process. Referring to Algorithm 1, there are at least two key points at which threads must cooperate. The first is the determination of scaling information, lines 1–5, which may be considered a separate scope to lines 6 forward. This task is readily solved using parallel reduction, a well known primitive. Turning attention to the main steps of the algorithm, lines 7–13 perform updates to matrix elements above the diagonal, specifically column j. By assigning the index of the loop at line 7 to thread index, increasingly more threads in this scope work as the outer loop progresses; a brief summary of this scope as executed in CUDA is detailed in Table 4.1. Within a warp, one may rely on SIMD execution, and thus updated column elements are available to threads of higher indices when needed.

Algorithm 1 LU decomposition with partial pivoting

Input : **A**, Batch of $N \times N$ matrices
Output: **A**, In–place LU decomposed matrices

```
1  for i ← 0 to N − 1 do
2  |   for j ← 0 to N − 1 do
   |   |   // find largest element q
3  |   end
4  |   scale[i] = 1.0/q;
5  end
6  for j ← 0 to N − 1 do
7  |   for i ← 0 to j − 1 do
8  |   |   sum = A[i][j];
9  |   |   for k ← 0 to i − 1 do
10 |   |   |   sum− = A[i][k] ∗ A[k][j];
11 |   |   end
12 |   |   A[i][j] = sum;
13 |   end
14 |   for i ← j to N − 1 do
15 |   |   sum = A[i][j];
16 |   |   for k ← 0 to j − 1 do
17 |   |   |   sum− = A[i][k] ∗ A[k][j];
18 |   |   end
19 |   |   A[i][j] = sum;
20 |   end
   |   // find index l of largest element q = scale[i]*fabs(sum)
21 |   if j != l then
   |   |   // swap rows j and l
   |   |   // update scale
   |   |   // save permutation details
22 |   end
23 |   if j! = N − 1 then
24 |   |   sum = A[j][j] ;
25 |   |   for k ← j to N − 1 do
26 |   |   |   A[i][j]/ = sum ;
27 |   |   end
28 |   end
29 end
```

As one might expect, matrices of side greater than a single warp require serialization of warp execution, due to the unpredictable way in which instructions are scheduled and dispatched within the Streaming Multiprocessor (SM), as illustrated in Fig. 4.1. Some parallelism is regained by mapping matrix to warp, for this scope alone.

No such limitations pervade lines 14–20, where loop index is also mapped to thread index, and column data is read from above the diagonal. Threads in this scope update from diagonal downwards; however, barrier synchronization is necessary before and after this scope. The particular column updated in a single iteration of

Table 4.1 Global memory read[], shared memory read(), write{}, critical† and arithmetic operations for several iterations and CUDA threads `t_id` of algorithm lines 7–14

k	t_id	j=2	j=3	j=4	j=5
–	1	(1,2)	(1,3)	(1,4)	(1,5)
0	1	−[1,0]*(0,2)	−[1,0]*(0,3)	−[1,0]*(0,4)	−[1,0]*(0,5)
–	1	{1,2}	{1,3}†	{1,4}†	{1,5}†
–	2		(2,3)	(2,4)	(2,5)
0	2		−[2,0]*(0,3)	−[2,0]*(0,4)	−[2,0]*(0,5)
1	2		−[2,1]*(1,3)†	−[2,1]*(1,4)†	−[2,1]*(1,5)†
–	2		{2,3}	{2,4}†	{2,5}†
–	3			(3,4)	(3,5)
0	3			−[3,0]*(0,4)	−[3,0]*(0,5)
1	3			−[3,1]*(1,4)†	−[3,1]*(1,5)†
2	3			−[3,2]*(2,4)†	−[3,2]*(2,5)†
–	3			{3,4}	{3,5}†
–	4				(4,5)
0	4				−[4,0]*(0,5)
1	4				−[4,1]*(1,5)†
2	4				−[4,2]*(2,5)†
3	4				−[4,3]*(3,5)†
–	4				{4,5}

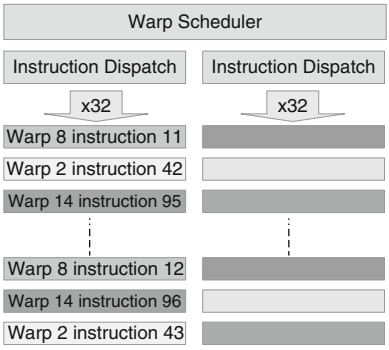

Fig. 4.1 An example of instruction scheduling and execution in a streaming multiprocessor

the outer loop is cached in shared memory before line 7, and written back to global after line 20. Shared memory buffers used for communication are declared using the `volatile` keyword, to ensure that write operations are not optimized out during compilation. Once the column update is complete, and working threads have written elements q before line 20 to another shared memory buffer, parallel reduction is employed in order to find the index of the pivot. Should the condition at line 21 be satisfied, then a row swap is completed by threads, storing temporary elements in registers. Thereafter, row elements are scaled by diagonal elements; once again loop index k is mapped to thread. Barrier synchronization is employed before the end of

Table 4.2 LU algorithm executed on K40c GPU device versus 16 Intel E5-2670 (Sandy Bridge) CPU threads

Batch size	Matrix size	K40c (s)	CPU(s)	mats./blk
800	256	1.5	1.5	1
1,600	128	0.33	0.45	1
8,000	64	0.20	0.30	2
16,000	32	0.05	0.11	4
64,000	16	0.03	0.15	8

the outer loop at line 29. An abbreviated listing of the main CUDA kernel is recorded in Appendix 1, based around the `float2` type, for processing complex data.

4.1.3 LU Results

An implementation of Algorithm 1 was written in C for execution on CPU, for use with row-major storage format matrices and complex (single precision) floating point data. This routine was compiled using a recent revision of the Intel compiler, with flags `-O3 -xHost` to ensure the highest degree of optimization, taking advantage of AVX hardware and instructions of the Sandy Bridge CPU. OpenMP was used to distribute matrices to separate threads for processing. The main GPU kernel as described and supporting routines including parallel reduction were compiled using `nvcc`, CUDA revision 5.5, for compute architecture 3.5 and with optimization flag `-O3`. Table 4.2 summarizes results, comparing execution times. Profiling using `nvvp` revealed a total global memory bandwidth of approximately 62 GB/s (54.5 GB/s read + 7.5 GB/s write). Both CPU and GPU routines were devoted to calculating the in-place LU decomposition alone. No permutations were stored; however, the sign of the permutation was recorded in memory, as is necessary for any subsequent calculation of matrix determinants. Crout's algorithm when executed on the K40c device experienced a 1.0–5.0x performance improvement over a single Sandy Bridge CPU socket, running 16 threads. The super-linear scaling of the CPU results was investigated further using tools from the Valgrind suite [8]. As expected, the effect had little correlation with cache performance; miss rates for both instructions and data were negligible for all matrix and batch sizes considered. However, profiling with `callgrind` did reveal that instructions devoted directly to the LU calculation itself steadily increased as a fraction of the total instructions, with matrix size. This fraction was as little as 60 % for a matrix of side 32, increasing to almost 100 % for matrices of side 256. Similarly, the percentage of instructions derived from other sources, particularly the Intel KMP interface for thread management and CPU affinity decreased to negligible contributions, for matrices of side 256.

4.2 QR Decomposition

While also a method that may be applied in the solution of systems of linear equations, the QR decomposition,

$$\mathbf{QR} = \mathbf{A}, \tag{4.4}$$

generally takes preeminence in a popular approach to eigendecomposition, the QR algorithm. In numerical implementations of the QR decomposition algorithm, the upper diagonal matrix R is constructed by the action of operations on A. R can be produced by one of several means, the most popular being Householder reflections, or Givens rotations [3]. This chapter focuses on the latter, whereby successive rotations G_i are applied, selectively eliminating elements below the diagonal of A, and producing the upper diagonal matrix R. One such step for the first column of a 3×3 complex matrix is illustrated in Eq. (4.5), where * denotes the complex conjugate.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s* & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a'_{21} & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix} \tag{4.5}$$

4.2.1 Theory

4.2.1.1 Serial QR Decomposition

The kernel of rotation matrix G_i is a 2×2 matrix that operates on pairs of values $a = a_{i,j}$ and $b = a_{i+1,j}$ in A, where elements c and s are chosen to eliminate the lower element in the operation:

$$\begin{bmatrix} c & s \\ -s* & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}. \tag{4.6}$$

Bindel et al. [1] give expressions for suitable c and s in a variety of contexts; the following are used in the remainder of this chapter for complex values, analogous to those for real values:

$$c = \pm \frac{|a|}{\sqrt{|a|^2 + |b|^2}}, \tag{4.7}$$

$$s = \pm \mathrm{sgn}(a) \frac{b}{\sqrt{|a|^2 + |b|^2}}, \tag{4.8}$$

where

$$\text{sgn}(a) = \begin{cases} a/|a| & \text{if } a \neq 0 \\ 1 & \text{if } a = 0 \end{cases}. \tag{4.9}$$

The concatenation of all orthogonal operations G_i comprises the transpose of the orthogonal matrix Q ie., using 0-based indexing,

$$Q^T A = \left[\prod_{j=0}^{j=N-2} \left\{ \prod_{i=j}^{i=N-2} G_i^j \right\} \right] A = R \tag{4.10}$$

where the superscript on G refers to the matrix column operated on during a particular iteration.

4.2.1.2 Parallel QR Decomposition

Sameh and Kuck [7] developed a parallel scheme dedicated to matrices of even side, in which the elimination process pictured in Eq. (4.5) can be carried in parallel across multiple rows and columns. Multiple independent Givens rotations $\tilde{Q}_{m,n}$ can be executed at the same time, where m and n refer to the row and column indices of the eliminated element. The product of these matrices constructs the matrix $\hat{Q}_i$, which is applied at the i-th step of the algorithm:

$$\hat{Q}_i = \prod \tilde{Q}_{m,n}. \tag{4.11}$$

For a given step i, the matrices $\tilde{Q}_{m,n}$ can be multiplied in any order to obtain $\hat{Q}_i$, as they are a direct sum of plane rotations [7]. As a result, $\hat{Q}_i$ is a block-diagonal matrix, with Givens rotations matrices G_i on the diagonal, as pictured in Eq. (4.12).

$$\hat{Q}_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_{k,l} & s_{k,l} & & & \\ & & & -s_{k,l} * & c_{k,l} & & & \\ & & & & & \ddots & & \\ & & & & & & c_{m,n} & s_{m,n} \\ & & & & & & -s_{m,n} * & c_{m,n} \\ & & & & & & & 1 \end{bmatrix} \tag{4.12}$$

The scheme from Sameh and Kuck is completed in $2N-3$ steps, where N is the rank of the matrix. The i-th transform is obtained by eliminating an entry in A at

the row m and column n, where m and n are given by

$$m = \begin{cases} \{N-i, N-i+1, \ldots, N-1-\delta(i)\} & 1 \leq i \leq N-1 \\ \{i-N+2, i-N+4, \ldots, N-1-\delta(i)\} & N \leq i \leq 2N-3 \end{cases}, \tag{4.13}$$

and

$$n = \begin{cases} \left\{1, 2, \ldots, \lceil \frac{i}{2} \rceil\right\} & 1 \leq i \leq N-1 \\ \left\{i-N+2, i-N+3, \ldots, \lceil \frac{i}{2} \rceil\right\} & N \leq i \leq 2N-3 \end{cases}, \tag{4.14}$$

with $\delta(i)$ defined as

$$\delta(i) = \begin{cases} 0 & i \text{ odd} \\ 1 & i \text{ even} \end{cases}. \tag{4.15}$$

Though other elimination patterns are possible, this approach has been proven to be one of the most efficient, both from a practical and mathematical point of view, as it is easy to implement and asymptotically optimal [2].

At each step of the process, the total number of rotations performed simultaneously, N_{rot}, is obtained by counting the total number of columns n and rows m affected:

$$N_{\text{rot}} = \begin{cases} \lceil i/2 \rceil & 1 \leq i \leq N-1 \\ \lceil i/2 \rceil - i + N - 1 & N \leq i \leq 2N-3 \end{cases}. \tag{4.16}$$

An example of the entries successively eliminated by this algorithm is shown in Fig. 4.2, for an 8×8 matrix. Numbers in the matrix correspond to the order in which the associated matrix element is eliminated in the algorithm.

$$\begin{bmatrix} * & * & * & * & * & * & * & * \\ 7 & * & * & * & * & * & * & * \\ 6 & 8 & * & * & * & * & * & * \\ 5 & 7 & 9 & * & * & * & * & * \\ 4 & 6 & 8 & 10 & * & * & * & * \\ 3 & 5 & 7 & 9 & 11 & * & * & * \\ 2 & 4 & 6 & 8 & 10 & 12 & * & * \\ 1 & 3 & 5 & 7 & 9 & 11 & 13 & * \end{bmatrix}$$

Fig. 4.2 Illustration of the successive elimination scheme in the QR parallel decomposition algorithm, for an 8×8 matrix

Algorithm 2 Outer loop of the parallel QR decomposition

Input: **A**, Batch of $N \times N$ matrices
Input: **NMAT**, Size of the batch
Input: **NMPBL**, Number of matrices to process per CUDA block
Input: **NTH**, Total number of CUDA threads per block
Input: **blocks**, CUDA Grid configuration
Output: **A,Q**, Upper diagonal batch of matrices R stored in place in A, batch of the transpose of the orthogonal matrices Q

```
1  blocks.x ← ⌈NMAT / NMPBL⌉
2  Q ← I_N
3  for 1 ≤ i ≤ 2*N-3 do
4      if i < N then
5          blocks.z ← ⌈i/2⌉
6      else
7          blocks.z ← ⌈i/2⌉ − i + N − 1
8      end
9      QR_Kernel <<< blocks,NTH >>> (Q,A,i)
10 end
```

4.2.2 GPU Implementation

The previous observations made in Sect. 4.1.2 related to global and shared memory accesses are also valid for the QR decomposition; therefore, each CUDA thread block is assigned one or more matrices to process, and N threads operate on a single matrix. The parallel QR algorithm is driven by an outer loop executed on the CPU, as detailed in Algorithm 2. This routine calculates the number of CUDA blocks to run in the x-dimension of the CUDA grid, initializes the orthogonal matrix Q as the identity matrix, and calculates the total number of Givens rotations that can be executed in parallel, based on Eq. (4.16). This number sets the z-dimension of the CUDA grid, to ensure that a total of N_{rot} Givens rotations are applied in parallel to the same matrix, at each iteration of the outer loop. Finally, each iteration launches the CUDA kernel to be executed on the GPU, shown in Algorithm 3. Each CUDA block in the x-dimension performs operations on multiple matrices A, and accumulates the results in the corresponding matrix Q. All threads first calculate the indices m, n of the entry to eliminate in their corresponding matrix. Threads then load rows $m-1$ and m, on lines 10 and 11, subsequently calculating their corresponding Givens rotation, on line 14. Algorithm 4 details this operation: multiple threads load the elements a and b defined in Eq. (4.6) through a shared memory broadcast on lines 1 and 2. The components of the Givens rotation kernel, c and s, are then evaluated on line 3 based on Eqs. (4.7) through (4.9). Turning attention back to Algorithm 3, the threads perform the Givens rotation on their

Algorithm 3 QR_KERNEL Core GPU kernel for the parallel QR decomposition

Input : **A**, Batch of $N \times N$ matrices
Input : **Q**, Batch of $N \times N$ matrices
Input : **NMPBL**, Number of matrices to process per block
Input : **i**, Kuck–Sameh iteration number
Input : **upperRow, lowerRow**, Shared memory buffers

```
   // Variables for convenience
1  tdx ← threadIdx.x
2  bdx ← blockIdx.x
3  bdz ← blockIdx.z

   // Calculate the location of the matrix on which to act
4  myMatrix ← tdx / N
5  matrixLocation ← ( bdx * NMPBL + myMatrix ) * N * N
6  myIndex ← tdx % N

   // Calculate the indices m and n on which to act
7  [m,n] ← CalcIndices (i,bdz)
   // Calculate the memory location of rows m and n
8  memLocUp ← matrixLocation + (m-1)*N+myIndex
9  memLocLo ← matrixLocation + m*N+myIndex

   // Load the rows m and m-1 of the matrix in the shared memory
10 upperRow[tdx] ← A[memLocUp]
11 lowerRow[tdx] ← A[memLocLo]
   // Wait for all the data to be loaded
12 syncthreads ()

   // Calculate the Givens rotation
13 smemIdx ← myMatrix*N + n
14 [c,s] ← CalcGivens(upperRow, lowerRow, smemIdx)

   // Apply the Givens rotation to all A matrices
15 ApplyGivens(A,upperRow,lowerRow,c,s,memLocUp,memLocLo)

   // Load the rows m and m-1 of Q in shared memory
16 upperRow[threadIdx.x] ← Q[memLocUp]
17 lowerRow[threadIdx.x] ← Q[memLocLo]

   // Accumulate the Givens rotation for all Q matrices
18 ApplyGivens(Q,upperRow,lowerRow,c,s,memLocUp,memLocLo)
```

corresponding matrix with the APPLYGIVENS routine. The details of this function are outlined in Algorithm 5. In the APPLYGIVENS routine, each thread within a CUDA block operates on a single matrix element of the two rows loaded in upperRow and lowerRow. The calculation presented in Eq. (4.6) is performed on lines 5 and 6. The threads then store the data back in place, in global memory,

Algorithm 4 CALCGIVENS Calculate the [c,s] values of a Givens rotation

Input: **upperRow, lowerRow**, Shared memory buffers
Input: **smemIdx**, Location of the corresponding matrix in the shared memory
Output: **c,s**, Givens rotation kernel values

```
// All threads calculate the Givens rotation for their
   corresponding matrix
1 a ← upperRow[smemIdx];
2 b ← lowerRow[smemIdx];
3 [c,s] ← Givens (a,b);
```

Algorithm 5 APPLYGIVENS Apply the [c,s] Givens rotation to an array of matrices

input : **M**, Batch of $N \times N$ matrices to update
input : **upperRow, lowerRow**, Shared memory buffers
input : **c,s**, Givens rotation kernel values
input : **memLocUp, memLocLo**, Per thread location to update in M

```
1 tdx ← threadIdx.x;
2 u ← upperRow[tdx];
3 l ← lowerRow[tdx];
  // Perform the rotation
4 syncthreads ();
5 upperRow[tdx] ← u*c+l*s;
6 lowerRow[tdx] ← u*(-s)+l*c;
  // Write the two modified rows back in global memory
7 M[memLocUp] ← upperRow[tdx];
8 M[memLocLo] ← (myIndex > n)*lowerRow[tdx];
```

on lines 7 and 8. Care is taken to introduce an exact zero for columns 1 through $n-1$ with the boolean condition `myIndex > n` on line 8, in order to avoid floating point approximations. The remainder of the Algorithm 3—lines 16 through 18—accumulates the rotations in the matrix Q. Note that the boolean condition on line 8 of Algorithm 5 does not apply to matrix Q, as can be discerned from the last line of `QR_Kernel` in Appendix 2.

Memory optimizations are included in the QR kernel implementation. A few constants, for example the current iteration number and the total batch size are stored in constant memory to provide fast data access. The bandwidth-cost of copying the data from the CPU to the GPU through a call to `cudaMemcpyToSymbol()` does not impact the overall performance of the algorithm. Care is taken to avoid non-coalesced global memory accesses by providing contiguous indices for global memory loads and stores.

Table 4.3 QR parallel decomposition algorithm executed on K40c GPU device versus 16 Intel E5-2670 (Sandy Bridge) CPU threads, in ms

	Batch size			
Matrix side	1,000	10,000	100,000	Matrices per block
16	1.370 (14.3)	7.475 (6.03)	68.14 (3.26)	64
32	6.732 (4.82)	55.76 (2.86)	534.0 (1.83)	32
64	48.70 (2.5)	457.9 (1.73)	4,630 (0.87)	16
128	404.9 (1.69)	4,025 (0.81)	–	8
256	3,172 (0.76)	32,151 (0.57)	–	4

The number in parenthesis indicates the speedup over the QR serial decomposition executed on the CPU

4.2.3 QR Results

A serial implementation of the QR decomposition algorithm as described in the first paragraph of Sect. 4.2.1 was written in C for execution on the CPU. The source code was compiled with the latest AVX optimizations available for Intel processors, with flags `-O3 -xHost`. The core GPU kernel `QR_Kernel` was compiled with the CUDA 5.5 revision of `nvcc` for compute architecture 3.5, and with `-O3` optimizations. The GPU method was tested on a Kepler K40c, while the CPU implementation was executed on a single Sandy Bridge CPU socket running 16 OpenMP threads. Benchmarking results are presented in Table 4.3. The GPU implementation of the QR algorithm as outlined here demonstrates a 0.6–14.3x performance improvement over a comparable CPU routine. The Nvidia profiler `nvvp` revealed a global memory bandwidth of 195 GB/s (97.5 GB/s read + 97.5 GB/s write).

Table 4.3 shows that the GPU results scale linearly at a constant matrix size. However, the scaling is not linear with the matrix size, at constant batch size; this effect can be attributed to a decreasing total number of matrices processed per block, as the size of the matrices increase. Therefore, more blocks are scheduled and executed on the GPU, resulting in a larger overhead. The QR GPU kernel as described was revealed to be *memory-bound* by the Nvidia profiler. Thus, additional optimizations to help the code scale with the matrix size may include increasing the total work performed by individual CUDA threads, in order to keep the total number of matrices processed per block constant. The super-linear behavior observed in the CPU scaling results was deduced to share similar origins as those of the CPU LU implementation.

4.3 Conclusion

This chapter has detailed new CUDA implementations of LU and QR decomposition, for large batches of matrices of side less than 1,024 elements. The kernels take advantage of several key GPU architectural features and display highly favorable performance and scaling as compared to comparable CPU implementations. However, QR decomposition was relatively more performant than LU decomposition, largely owing to the need for warp serialization and fairly excessive synchronization in the latter. Performance for initial kernels was improved significantly through introduction of several techniques guided by profiling. These techniques included configuring cache and shared memory in software, as well as optimizing thread blocksize and shared memory buffer size. Further optimizations and alternative kernels for these important methods are the subjects of ongoing work.

Acknowledgements The authors would like to thank Muhammed Kabiru Hassan and Sreejith Ganesh Jaya for bringing applications to their attention that benefit from the routines detailed here. The authors are also very grateful to reviewers from Nvidia for their comments and improvements to the manuscript.

Appendix 1

```
__global__ void luDecomposition ( float2 * inputMatrices, float * devSign ){

// NOTES:
// array indices have been simplified for readability
// eg.,
// #define buf_index       ( vectorIndex + myMatrix * MATRIX_SIDE )
// common tasks have been relegated to device functions
// temporary variables
float2 sum,dum,tmp,tmpr,tmpl;

// scratch space
__shared__ float               sign [ NUM_MATRICES ];
__shared__ volatile float      scale [ NUM_MATRICES * MATRIX_SIDE ];
__shared__ volatile float2     reduce [ NUM_MATRICES * MATRIX_SIDE ];
__shared__ volatile float2     vectors [ NUM_MATRICES * MATRIX_SIDE ];
__shared__ volatile int        indices [ NUM_MATRICES * MATRIX_SIDE ];
```

```
// index to matrix for processing
int myMatrix    = threadIdx.x / MATRIX_SIDE;

// index to vector for processing
int vectorIndex = threadIdx.x % MATRIX_SIDE;

// which warp
int myWarp      = vectorIndex / 32;


// initialize permutation signs
sign[threadIdx.x % NUM_MATRICES]=1.0f;

// initialize shared memory
initFloat2Buffer( vectors, FLOAT_MIN );


// determine scaling information
for (int i=0; i < MATRIX_SIDE; ++i){

        __syncthreads();

        // load shared memory

        vectors [ buf_index ].x = inputMatrices [ row_i_index ].x;
        vectors [ buf_index ].y = inputMatrices [ row_i_index ].y;

        __syncthreads();

        // find maxima by reduction
        findVectorMaxima ( vectors, vectorIndex, myMatrix );

        __syncthreads();
        // write scaling information
        if ( vectorIndex ==i ){

                // should test for singular
                scale [ scale_index ] = abs ( vectors [ buf_00 ].x );
        }
}

// initialize shared memory
initFloat2Buffer ( vectors, 0.0f );

for (int j=0; j<MATRIX_SIDE; j++){

        __syncthreads();

        // load the j column to shared
        vectors [ buf_index ].x         = inputMatrices [ col_j_index ].x;
        vectors [ buf_index ].y         = inputMatrices [ col_j_index ].y;

        __syncthreads();

        myMatrix=myWarp;
        if (( myMatrix < NUM_MATRICES ) && ( vectorIndex < j)){
         for (int i=0; i<WARPS_PER_MATRIX; i++){


                        sum.x = vectors [ buf_index ].x;
                        sum.y = vectors [ buf_index ].y;

                        for (int k=0; k< MATRIX_SIDE ; k++){

                                if (k>=vectorIndex) break;

                                tmpl = inputMatrices [ col_k_index ];

                                tmpr.x = vectors [ buf_k ].x;
                                tmpr.y = vectors [ buf_k ].y;

                                sum.x -= (tmpl.x * tmpr.x - tmpl.y * tmpr.y);
                                sum.y -= (tmpl.y * tmpr.x + tmpl.x * tmpr.y);

                                vectors [ buf_index ].x = sum.x;
                                vectors [ buf_index ].y = sum.y;

                        }

        }
}
myMatrix = threadIdx.x / MATRIX_SIDE;

__syncthreads();
if ((vectorIndex >=j) && (vectorIndex < MATRIX_SIDE)){

        sum.x = vectors [ buf_index ].x;
        sum.y = vectors [ buf_index ].y;
```

```
                for (int k=0; k< j; k++){

                        tmpl = inputMatrices [ col_k_index ];

                        tmpr.x = vectors [ buf_k ].x;
                        tmpr.y = vectors [ buf_k ].y;

                        sum.x -= (tmpl.x * tmpr.x - tmpl.y * tmpr.y);
                        sum.y -= (tmpl.y * tmpr.x + tmpl.x * tmpr.y);
                        vectors [ buf_index ].x= sum.x;
                        vectors [ buf_index ].y= sum.y;
                }
        }
        __syncthreads();

        // write j column back to global
        inputMatrices [ col_j_index ].x = vectors [ buf_index ].x;
        inputMatrices [ col_j_index ].y = vectors [ buf_index ].y;

        // initialize shared memory
        initFloat2Buffer ( reduce, FLOAT_MIN );

        __syncthreads();

        if (vectorIndex >= j){

                // init for pivot search by reduction
                reduce [ buf_index - j ].x = abs ( vectors [ buf_index ].x ) \
                    /  scale [ scale_index ];
                indices [ buf_index - j ]  = vectorIndex;
        }

        __syncthreads();
        findVectorMaximaKey ( reduce, indices, vectorIndex, myMatrix );
        __syncthreads();

        // possible row swap
        if (j != indices [ buf_00 ]){

                int i = indices [ buf_00 ];

                // each thread swaps one row element with another row element

                sum                             = inputMatrices [ row_i_index ];
                inputMatrices [ row_i_index ]   = inputMatrices [ row_j_index ];
                inputMatrices [ row_j_index ]   = sum;

                if (vectorIndex==0){
                        scale [ buf_i ]         = scale [ buf_j ];
                        sign [ myMatrix ]       *= -1.0f;
                }
        }

        __syncthreads();

        // final scaling
        if ( j != MATRIX_SIDE-1){

                dum = inputMatrices [ diag_j_index ];

                if (vectorIndex >= j+1){

                        tmp                             = inputMatrices [ col_j_index ];
                        tmp                             = divide ( tmp, dum );
                        inputMatrices [ col_j_index ]   = tmp;
                }
        }

        __syncthreads();

    }// end j loops

    // write out sign
    if (vectorIndex == 0) devSign [ sign_ind ] = sign [ myMatrix ] ;


}
```

Appendix 2

```
// Iteration number and total batch size are stored in constant memory
__device__ __constant__ int cmem_k, cmem_size;

__global__ void QR_Kernel(float2 *matrices, float2 *q_complete) {

        //// Shared memory buffer rows
        // NOTE: This kernel as presented does not safeguard against buffer under/overflow
        __shared__ float2 upper_row[NMPBL*MATRIX_SIDE];
        __shared__ float2 lower_row[NMPBL*MATRIX_SIDE];

       //// Convenience indices
       // Index to matrix for processing
       int myMatrix  = threadIdx.x / MATRIX_SIDE;
       // Index to vector for processing
       int vectorIndex  = threadIdx.x % MATRIX_SIDE;
       // Matrix offset for this block
       unsigned int memoryStride        = \
            ( blockIdx.x * NMPBL + myMatrix  ) * MATRIX_SIDE * MATRIX_SIDE ;

       //// Set column and line number we want to eliminate
       int my_i, my_j = 0; int iter = cmem_k;

       int dk = delta_k(iter);
       if( iter < MATRIX_SIDE ) {
              my_j = blockIdx.z;
              my_i = (MATRIX_SIDE-iter) + 2*my_j;
       } else {
              my_j = (iter-MATRIX_SIDE) + blockIdx.z + 1;
              my_i = (iter-MATRIX_SIDE) + 2*blockIdx.z + 2;
       }

       //// Load row data - if condition avoids out of bounds accesses
       if(memoryStride + my_i*MATRIX_SIDE + vectorIndex < cmem_size_kuck*MATRIX_SIDE*MATRIX_SIDE) {
              upper_row[threadIdx.x] = matrices[memoryStride + (my_i-1)*MATRIX_SIDE + vectorIndex];
              // Lower row w/ leading zero after rotation
              lower_row[threadIdx.x] = matrices[memoryStride + my_i*MATRIX_SIDE + vectorIndex];
       }

       //// Wait for all the data to be loaded first
       __syncthreads();

       float2 u,v,c,s;
       float f,g,den;

       u = upper_row[myMatrix*MATRIX_SIDE + my_j]; // broadcast operation from SMEM
       v = lower_row[myMatrix*MATRIX_SIDE + my_j]; // broadcast operation from SMEM

       //// Calculate c and s
       f = u.x*u.x + u.y*u.y;
       g = v.x*v.x + v.y*v.y;

       // Algorithm is provided in [BDK02]
       if( g < 2e-16 ) {

              c.x = 1.0f; c.y = 0.0f;
              s.x = 0.0f; s.y = 0.0f;

       } else if (f< 2e-16) {

              c.x = 0.0f; c.y = 0.0f;

              // s = conj(v)/g
              den = 1.0f/g;
              s.x = v.x*den; s.y = -v.y*den;

       } else {
              // r = sqrt(f + g)
              den = rsqrt(f + g);
              // c = f/r
              c.x = sqrt(f)*den; c.y = 0.0f;

              // s = x/f * conj(y) / r
              // den = -1/(f*r)
              den *= rsqrt(f);

              s.x = (u.x*v.x + u.y*v.y)*den;
              s.y = (u.y*v.x - u.x*v.y)*den;
       }

       //// Compute the two rows update
       // u*c + v*s
       // Load data
       u = upper_row[threadIdx.x];
       v = lower_row[threadIdx.x];
       // Perform product: real part
       f = (u.x*c.x - u.y*c.y) + (v.x*s.x - v.y*s.y);
```

```
        // Perform product: imaginary part
        g = (u.x*c.y + u.y*c.x) + (v.x*s.y + v.y*s.x);

        float2 tmp;
        tmp.x = f; tmp.y = g;

        // u*-conj(s) + v*c
        // Perform product: real part
        f = -(u.x*s.x + u.y*s.y) + (v.x*c.x - v.y*c.y);
        // Perform product: imaginary part
        g = (u.x*s.y - u.y*s.x) + (v.x*c.y + v.y*c.x);

        ///// Store: synchronization is necessary to avoid overwriting data ...
        ///// ...for warps that are still getting data from the broadcast operation
        __syncthreads();
        upper_row[threadIdx.x] = tmp;

        lower_row[threadIdx.x].x = f;
        lower_row[threadIdx.x].y = g;

        //// Write data in the original matrix
        if(memoryStride + my_i*MATRIX_SIDE + vectorIndex < cmem_size*MATRIX_SIDE*MATRIX_SIDE) {
               matrices[memoryStride + (my_i-1) * MATRIX_SIDE + vectorIndex ] = upper_row[threadIdx.x];

               matrices[memoryStride + my_i * MATRIX_SIDE + vectorIndex ].x = \
                     (vectorIndex > my_j) * lower_row[threadIdx.x].x;
               matrices[memoryStride + my_i * MATRIX_SIDE + vectorIndex ].y = \
                     (vectorIndex > my_j) * lower_row[threadIdx.x].y;
        }

        //// Load rows of Q to be updated
        if(memoryStride + my_i*MATRIX_SIDE + vectorIndex < cmem_size*MATRIX_SIDE*MATRIX_SIDE) {
               upper_row[threadIdx.x] = q_complete[memoryStride + (my_i-1) * MATRIX_SIDE + vectorIndex ];
               lower_row[threadIdx.x] = q_complete[memoryStride + my_i * MATRIX_SIDE + vectorIndex ];
        }

        //// Apply the Givens rotation
        u = upper_row[threadIdx.x];
        v = lower_row[threadIdx.x];

        // Q[i-1,k] = C*Q[i-1,k] + S*Q[i,k]
        // Perform product: real part
        f = (u.x*c.x - u.y*c.y) + (v.x*s.x - v.y*s.y);
        // Perform product: imaginary part
        g = (u.x*c.y + u.y*c.x) + (v.x*s.y + v.y*s.x);

        tmp.x = f; tmp.y = g;

        // Q[i,k] = -S'*Q[i-1,k] + C*Q[i,k]
        // Perform product: real part
        f = -(u.x*s.x + u.y*s.y) + (v.x*c.x - v.y*c.y);
        // Perform product: imaginary part
        g = (u.x*s.y - u.y*s.x) + (v.x*c.y + v.y*c.x);

        // No synchronization necessary here;
        // each thread operates independently on a single matrix element
        upper_row[threadIdx.x] = tmp;

        lower_row[threadIdx.x].x = f;
        lower_row[threadIdx.x].y = g;

        //// Write to global
        if(memoryStride + my_i*MATRIX_SIDE + vectorIndex < cmem_size*MATRIX_SIDE*MATRIX_SIDE) {
               q_complete[memoryStride + (my_i-1) * MATRIX_SIDE + vectorIndex ] = upper_row[threadIdx.x];
               q_complete[memoryStride + my_i * MATRIX_SIDE + vectorIndex ] = lower_row[threadIdx.x];
        }
}

// CPU driver loop
extern "C"{
        blocks.x = (int)ceil((float)size/(float)NMPBL);
        cudaMemcpyToSymbol(cmem_size,&size,sizeof(size));

        // Set the shared memory bank size to 8 bytes / 64 bits
        cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte);

        for(int k = 1;k<=2*MATRIX_SIDE-3;k++) {
               // Calculate the total number of rotations to apply at once
               // Launch blocks.z additional rows of CUDA blocks to compute the Nrot rotations
               if( k < MATRIX_SIDE ) {
                     blocks.z = (int)ceil((float)k/2.0f);
               } else {
                     blocks.z = (int)ceil((float)k/2.0f) - k + MATRIX_SIDE-1;
               }
               // Update constant memory
               cudaMemcpyToSymbol(cmem_k,&k,sizeof(k));
               // Launch the main kernel;
               // calculates the Givens rotations and places them in the temporary matrix Q_A
               QR_Kernel <<< blocks,NTH >>> (matrices,q_complete);
        }//end main loops
}
```

References

1. Bindel, D., Demmel, J., Kahan, W., Marques, O.: On computing givens rotations reliably and efficiently. ACM Trans. Math. Softw. **28**(2), 206–238 (2002)
2. Cosnard, M., Robert, Y.: Complexity of parallel QR factorization. J. Assoc. Comput. Machinery **33**, 712–723 (1986)
3. Golub, G.H.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
4. Lucente, E., Monorchio, A., Mittra, R.: An iteration-free MoM approach based on excitation independent characteristic basis functions for solving large multiscale electromagnetic scattering problems. IEEE Trans. Antennas Propag. **56**(4), 999–1007 (2008)
5. Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: Numerical Recipes in C: The Art of Scientific Computing, 2nd edn. Cambridge University Press, Cambridge (1993)
6. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM, Philadelphia (2003)
7. Sameh, A.H., Kuck, D.J.: On stable parallel linear system solvers. J. Assoc. Comput. Machinery **25**, 81–91 (1978)
8. Seward, J., Nethercote, N., Weidendorfer, J.: Valgrind 3.3: Advanced Debugging and Profiling for GNU/Linux Applications. Network Theory Ltd., Bristol (2008)
9. Sreejith, G.J., Jolad, S., Sen, D., Jain, J.K.: Microscopic study of the $\frac{2}{5}$ fractional quantum Hall edge. Phys. Rev. B **84**, 245104 (2011)
10. Sreejith, G.J., Toke, C., Wójs, A., Jain, J.K.: Bipartite composite fermion states. Phys. Rev. Lett. **107**, 086806 (2011)