AN IMPLEMENTATION OF THE QR DECOMPOSITION OF REAL MATRICES ON A GRAPHICS PROCESSING UNIT

UWE KÖCHER AND ALEXANDER PAPROTNY

Abstract. Graphic processing units provide a low-cost parallel computing architecture. Only recently emerged their employment in scientific computation as an area of research. We present an implementation of the QR decomposition on the NVDIA GeForce 8800 GT unit using the CUBLAS library made available by the manufacturer and analyze its performance.

Contents

1. Introduction 2
2. Computing a QR decomposition via Householder transformations 2
2.1. Algorithms 2
3. CUDA programming 7
4. Performance analysis 8
4.1. Strategies 8
4.2. Technical data 8
4.3. Empirical block size optimization 8
4.4. Comparison of GPU and CPU implementations 9
5. Conclusions 9
6. Open problems 9
References 14

Key words and phrases. General purpose computation on graphics processing units, QR decomposition, partitioned algorithm, parallel computing, GPGPU.

1. Introduction

As one commonly observes, the growth of problem sizes is a trend in scientific computing. This results in a demand on hardware architectures which accelerate the various computations and appropriate parallel algorithms. As dedicated parallel computer systems are too expensive to achieve a vast popularity, graphics processing units[1] provide a low-cost alternative. However, there are as yet many open problems concerning implementations of numerical algorithms on these units under exploiture of their architecture. We have implemented an algorithm for QR decomposition of real matrices on the NVDIA GeForce 8800GT using the yet available BLAS-routines. Our approach is based on a partitioned algorithm described in [1]. We also considered a recursive version from the same paper, which shows an astonishing performance when implemented in MATLAB. Our attempts with the latter one, however, failed with respect to numerical stability. It was also part of our proceeding to re-implement the algorithm on an CPU with respective BLAS routines. In the beginning, we tested the algorithms in MATLAB and subsequently used the CUDA/CUBLAS and MKL-BLAS interfaces respectively in programming language C. This paper is outlined as follows: In the first few sections, we briefly analyze the mathematical background and discuss the chosen strategies and algorithms. Afterwards, we give a summary of our experience during the implementation. Eventually, we investigate the results of runtime and performance measurements with special focus on comparison to an implementation on a common CPU.

2. Computing a QR decomposition via Householder transformations

2.1. Algorithms.

2.1.1. *Basic procedures.*

Definition 1. *QR decomposition Let $A \in \mathbb{R}^{(m,n)}$, $m \geq n$, $Q \in \mathbb{R}^{(m,m)}$ be orthogonal and $R \in \mathbb{R}^{(n,n)}$ upper triangular. if*

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} \tag{2.1}$$

then (Q, R) is called a QR decomposition of A.

In a householder step we aim to reflect the first column of a matrix on the direction of the first unit vector. Therefore we reflect with respect to an appropriate hyperplane in $\mathbb{R}^n$. Repeating this step recursively, we obtain a QR decomposition of A in the following manner:

$$\underbrace{H_n \cdot \ldots \cdot H_2 \cdot H_1}_{Q^T} A = \begin{pmatrix} R \\ 0 \end{pmatrix} \quad \Longleftrightarrow \quad A = \underbrace{H_1 \cdot H_2 \cdot \ldots \cdot H_n}_{Q} \begin{pmatrix} R \\ 0 \end{pmatrix} \tag{2.2}$$

$$H_1 A = H_1 \left(\begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} \quad A(:,2:\text{n}) \right) = \left(\begin{bmatrix} r_{11} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \begin{array}{c} R(1,2:\text{n}) \\ \hline \tilde{A}(2:\text{m},2:\text{n}) \end{array} \right), \tag{2.3}$$

where the *Householder matrix* H is defined as

[1] in the following referred to as GPU

$$H := I_m - \beta v v^T, \beta := \frac{2}{v^T v}, \tag{2.4}$$

where $v \in \mathbb{R}^m$ denotes an appropriate Hyperplane in $\mathbb{R}^m$ and $I_m \in \mathbb{R}^{(m,m)}$ the identity mapping in $\mathbb{R}^m$.

So rarely is an explicit representation of Q needed in applications, that we can restrict ourselves to the storage of the v vectors along the computation. Scaling the first component of v to 1 enables us to store the v vectors efficiently in the R matrix (c.f. later).

A transformation in the manner of (2.3) is obtained by a *Householder vector*

$$v := a_1 \pm ||a_1||_2 e^1, \tag{2.5}$$

where e^k denotes kth unit vector in $\mathbb{R}^m$:

$$e^k(i) = \begin{cases} 1, & i = k \\ 0, & else \end{cases}$$

If $a_{11} \approx 1$, using subtraction in (2.5) would cause severe cancellation. In order to avoid this we replace (2.5) by[2]

$$v := a_1 + \text{sign}(a_{11}) ||a_1||_2 e^1 \tag{2.6}$$

Unfortunately, we obtain negative elements in the entire triangular matrix due to

$$\begin{aligned} v := a_1 \pm ||a_1||_2 e^1 \quad &\Rightarrow \\ Ha_1 = \left(I_m - 2 \frac{v\ v^T}{v^T v} \right) a_1 = &\mp ||a_1||_2\ e^1 \end{aligned}$$

In [2] Parlett's formula is applied to circumvent this in a numerically stable way:

$$x_1 > 0 \quad \Rightarrow \quad v_1 := x_1 - ||x||_2 = \frac{x_1^2 - ||x||_2^2}{x_1 + ||x||_2} = \frac{-(x_2^2 + \ldots + x_n^2)}{x_1 + ||x||_2} \tag{2.7}$$

$$x_1 \le 0 \quad \Rightarrow \quad v := x - ||x||_2 \quad e^1 \tag{2.8}$$

[2] c.f.[4]

We summarize these reasonings in the following

Algorithm 1. *Householder vector using Parlett's formula*
function *[v, β] = house(x)*

$n :=$ ***length***(x); $\sigma := x(2{:}n)^T\ x(2{:}n)$; $v := \begin{bmatrix} 1 \\ x(2:n) \end{bmatrix}$

if $(\sigma = 0)$
$\beta := 0$
else
$\mu := \sqrt{x(1)^2 + \sigma}$
if $(x(1) \leq 0)$
$v(1) := x(1) - \mu$
else
$v(1) := \frac{-\sigma}{x(1)+\mu}$
end
$\beta := \frac{2v(1)^2}{\sigma+v(1)^2}$
$v := \frac{v}{v(1)}$
end

The algorithm (1) is performed within $3n$ flops (c.f. [2], pp.210f.). It is recommended to normalize x in advance to avoid overflow. This costs additional n flops. The update step is performed by virtue of

$$HA = (I_m - \beta vv^T)A = A - \beta v(v^T A) \tag{2.9}$$

Here we expend mn flops for the outermost subtraction, $2mn$ flops for the first matrix-vector-multiplication and eventually $2mn$ flops for the remaining outer product and scaling by β. This sums up to $5mn$ flops. Note that the order of sequence of operations must be considered. Otherwise, the complexity will be affected. Choosing

$$HA = (I_m - \beta vv^T)A = A - ((\beta(vv^T))A)$$

would raise the complexity to $2m^2n + \mathcal{O}(m^2)$ flops. As one can observe, a setting with $m >> n$ dramatically contribute to a high expense.

In the kth step, of QR decomposition, a householder vector $v \in \mathbb{R}^m, m \geq n \geq k \geq 1$ has structure

$$v^{(k)} = (\underbrace{0, \ldots, 0}_{k-1}, 1, v_{k+1}^{(k)}, \ldots, v_m^{(k)})^T \tag{2.10}$$

As stated above, computing Q in each step is utterly inefficient. Hence, storing $v^{(k)}$ suffices. This enables us to store store the R matrix and v vectors in entirely in the memory allocated for A occupying some additional memory for a vector containing the β values.

$$A = \begin{pmatrix} r_{11} & r_{12} & \cdots & & r_{1n} \\ v_2^{(1)} & r_{22} & & & r_{2n} \\ v_3^{(1)} & v_3^{(2)} & \ddots & & \vdots \\ \vdots & & \ddots & r_{n-1,n-1} & r_{n-1,n} \\ v_n^{(1)} & v_n^{(2)} & \cdots & v_n^{(n-1)} & r_{nn} \\ v_{n+1}^{(1)} & v_{n+1}^{(2)} & \cdots & & v_{n+1}^{(n)} \\ \vdots & & & & \vdots \\ v_m^{(1)} & v_m^{(2)} & \cdots & & v_m^{(n)} \end{pmatrix}, \beta := \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix} \tag{2.11}$$

We refer to (2.11) as **VRβ decomposition**.

With regard to an implementation on a GPU, this shows that the matrix A can be written to the memory of the GPU (provided that there is enough memory available) and the result, i.e. the VRβ decomposition, can be kept directly on the device. After each Householder step one can additionally source out the first line and column, which are no longer needed for the further computation. So one can transfer the VRβ-result to a different memory device simultaneously to further computation.

2.1.2. *Blocked algorithm.* In each step of QR decomposition an orthogonal Householder matrix is generated, which is a rank-1-modification of the identity:

$$H_k = \begin{pmatrix} I_{k-1} & \\ & I_{m-k+1} - \beta \left(v^{(k)}\left(k:m\right)\right)\left(v^{(k)}\left(k:m\right)\right)^T \end{pmatrix}$$

$$Q = H_1 \cdot \ldots \cdot H_n$$

The following relation enables us to update the Q matrix efficiently employing BLAS-3 routines.

Lemma 1. *Let $Q = I_m + WY^T \in \mathbb{R}^{(m,m)}$ be orthogonal, where $W, Y \in \mathbb{R}^{(m,j)}$.*

Defining $H := I_m - \beta vv^T$ with $v \in \mathbb{R}^m$ and $z := -\beta Qv$, we obtain the equation

$$\tilde{Q} = QH = I + \tilde{W}\tilde{Y}^T$$

with $\tilde{W} = [W\ z]$ and $\tilde{Y} = [Y\ v]$, where $\tilde{W}, \tilde{Y} \in \mathbb{R}^{(m,j+1)}$.

Proof.

$$\begin{aligned} QH &= \underbrace{\left(I_m + WY^T\right)}_{Q}\underbrace{\left(I_m - \beta vv^T\right)}_{H} \\ &= I + WY^T - \beta Qvv^T \\ &= I + WY^T + zv^T \\ &= I + [W\ z]\,[Y\ v]^T \end{aligned}$$

□

Notice that in each step of the sequence $Q_0, Q_1, ..., Q_n$ generated by the algorithm, where

$$Q_0 := I_m,$$
$$Q_i := Q_{i-1} H_i, \quad 1 \leq i \leq n,$$

the Q_i matrices can be updated by applying lemma (1) inductively. This gives rise to

Algorithm 2. *Block representation of Q*
function *[W,Y] = createWY(V, β)*
  $Y = v^{(1)}$
  $W = -\beta_1 v^{(1)}$
  for $i = 2 : n$
    $z = -\beta_i \left(v^{(i)}(i : m) + W \left(Y^T(:, i : m)\ v^{(i)}(i : m) \right) \right)$
    $W = [W\ z]$
    $Y = \left[Y\ v^{(i)} \right]$
  end

This algorithm requires $2n^2(m - \frac{n}{3})$ flops, provided that the zeros in $v^{(i)}$ are exploited (c.f.[2] for a detailed discussion). Note that again an explicit computation of Q is unfavorable. Instead, one can efficiently update the matrix using the W and Y representations by virtue of

$$C \leftarrow Q^T C = (I_m + WY^T)^T C = C + Y(W^T C)$$

In the blocked version of the VR-beta decomposition we compute VR-beta of r columns. Afterwards the remaining part of the matrix is updated. First, we compute W and Y. Subsequently, we update A by

$$A\,(1 : m, r : n) = A\,(1 : m, r : n) + Y \left(W^T A\,(1 : m, r : n) \right)$$

Afterwards we restart the algorithm on $A\,(r : m, r : n)$ (c.f. [2], p.226).

2.1.3. *Recursive algorithm.* We considered another QR-Householder algorithm by Åke Björk. It is a pretty fast recursive QR algorithm (especially in the non quadratic case). If you want to compute (explicit) Q and R with MATLAB this algorithm is superior with respect to runtime. Unfortunately, we failed to implement it in a numerically stable way on GPU. Either, this is simply an fault of ours or a result of the non-standard single precision computation on the device, which is not conform to the IEEE754 single precision standard. Because of that fact, we are only present the the results for the last algorithm. The procedure and MATLAB code to this recursive QR-algorithm can be found in "The calculation of linear least squares problems" [1].

3. CUDA programming

There are basically 3 levels of CUDA programming:

- Level 1 - CUDA libraries like cublas
- Level 2 - CUDA runtime
- Level 3 - CUDA driver (kernel programming)

Our implementation is situated on the CUBLAS level.
There were 2 main types of problems during implementation:

- Missing cublas functions
- Understanding the LDA parameter, which can be confusing to someone who is not familiar with BLAS interfaces.

Not all common BLAS functions are available from the CUBLAS library. A problem occurred when we were trying to implement

$$A = \left(I_m + \beta v v^T\right) A = A + \beta v \left(v^T A\right)$$

To implement $v^T A$ we couldn't use a CUBLAS function directly. Instead, we used

$$v^T A = \left(A^T v\right)^T =: z^T$$

Now we could employ cublasSgemv (generic matrix-vector product). Fortunately, z^T was needed for the following rank-1 update:

$$A = A + \beta v z^T.$$

"LDA" abbreviates "Leading dimension of A" and refers to the distance of two columns in the memory. It is simply the number of rows in the original matrix if one works with submatrices. Since a matrix A is stored row-wise, the distance of two columns of a submatrix $A(r : m, r : n)$ deviates from the number of its rows.

All algorithms are available in MATLAB, CUDA/cublas and Intel MKL.
If you like to see the code ask our supervisor Cem Bassoy (bassoy@tuhh.de).

Versions:

- (VR, b) = qrhousenb(A) - non blocked VR-beta version
- (Q, R) = qrhousenb2(A) - non blocked VR-beta version
- (VR, b) = qrhouseb(A, r) - blocked VR-beta version
- (Q, R) = qrhouseb2(A, r) - blocked VR-beta version
- (Y,T,R) = recqr(A) - recursive QR-Householder
- (Q, R) = recqr2(A) - recursive QR-Householder (this is pretty fast in MATLAB)

4. Performance analysis

4.1. **Strategies.** We restrict ourselves to the blocked algorithm QRHOUSEB. So severely increases the numerical error of RECQR with the number of processed columns, that it cannot be considered competitive with MATLAB implementation w.r.t. stability. Numerically stable implementation of RECQR based on CUBLAS or MKLBLAS respectively has not yet been found by us, which gives rise to a new research problem for graduate student. For all test cases, we had random matrices with independent columns generated. Our benchmark consisted of

(1) square matrices up to size 4096 by 4096.
(2) rectangular matrices with 128, 256, 512, 1024, 2048 columns and variable number of rows

As a first step, we determined optimal block size for either implementation. Due to the hardware architectures the domain of parameters for block size has been restricted to powers of 2. During the optimization, we varied block sizes from 8 up to 512 for each of the cases enumerated above. Subsequently, we performed our benchmark separately for *VRbeta* decomposition and entire QR decomposition, i.e. VRbeta followed by explicit computation of Q. The former one shows an asymptotic complexity of $\mathcal{O}(mn^2)$, whereas the latter computation is in $\mathcal{O}(m^3 + mn^2)$. Interleaving the computation of Q dramatically reduces the expense on memory and enables a larger test case scope. Apart from that, we are not aware of any application of QR decomposition which requires an explicit representation of Q except a rather trivial one, namely the completion of one single vector to an orthonormal basis in n space. As far as possible, memory allocations and initializations have been excluded from measured runtimes. In order to achieve a meaningful comparison, runtime and performance tests have been performed in respective optimal block size.

4.2. **Technical data.** Here is a brief description of our hardware environment:
GPU:

- NVDIA GeForce 8800 GT
- 128 parallel processing units each with word length of 16 bit
- 512 MB memory on board

CPU:

- Intel Core 2
- 2 x 2.4 GHz clock frequency
- 1.98 GB RAM

4.3. **Empirical block size optimization.** Fig. 1 shows measurement of runtimes for different block sizes (labeled by r) and numbers of matrix rows and columns on the GPU. From these data we inferred an optimal size of $r = 16$. The observable oscillatory behavior can be smoothened by padding, which is yet another open problem. We consider it memorable that for values above $r = 16$, the order of the plots by runtime varies with the number of columns n.

Fig.2 depicts the plots of the results of the same benchmarks on the CPU. One perceives a evidently more noisy behavior an far less distinct plots, which results from the operating system's scheduling. We determined the optimal size at $r = 32$.

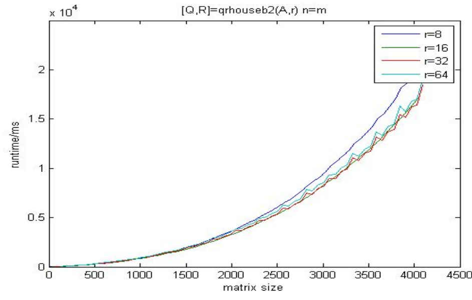
4.4. **Comparison of GPU and CPU implementations.** As shown in Fig., the explicit QR decomposition on GPU has a remarkable speedup of about 3.5. Unfortunately, our implementation of VRbeta decomposition (Fig.) does not yield any speedup but rather decelerates the computation in comparison to MKL version. This results from the fact that BLAS routines on GPU on levels less than 3 are not competitive against MKL with respect to small dimensional data items and that some awkward initialization subroutines are inevitable during the computation. This suggest the approach to outsource the entire non-blocked-subroutine to the CPU and restrict utilization of the GPU to matrix update operations. Regretfully, even this approach turned out to be futile: So severe is the overhead caused by data transmission operations entailed by the hybrid implementation that the original execution times are even tripled.

5. Conclusions

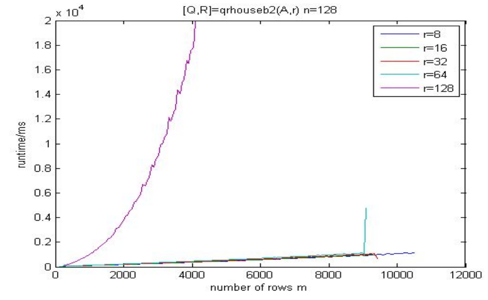
- QR decomposition can be computed efficiently on GPU using the blocked algorithm.
- Beyond a certain threshold problem size, the GPU implementation is superior with respect to runtime and performance
- A speedup of about 3.5 has been attained
- An efficient combination of GPU and CPU operations is difficult due to expensive copy operations

6. Open problems

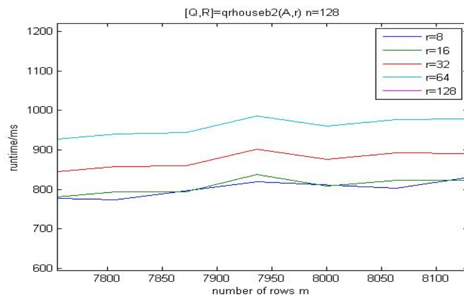
- appropriate padding method in order to avoid overhead on GPU and to smoothen oscillatory behavior of runtimes
- efficient CPU/GPU-hybrid implementation of VR-beta decomposition circumventing copy operations
- numerically stable implementation and investigation of recursive algorithm



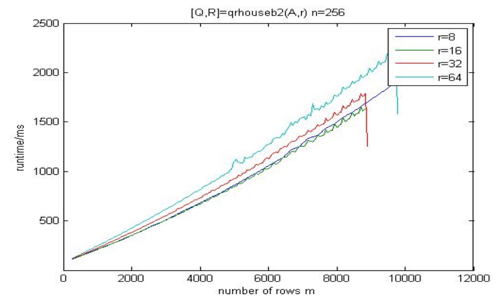
(a) Varying block sizes r with square matrices



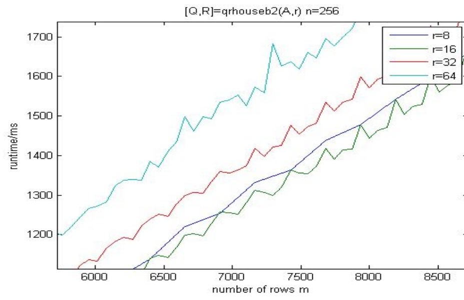
(b) Varying block sizes r with n = 128



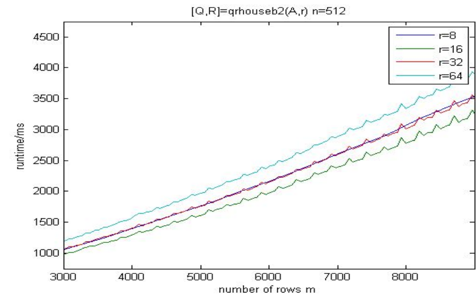
(c) Varying block sizes r with n = 128 with zoom



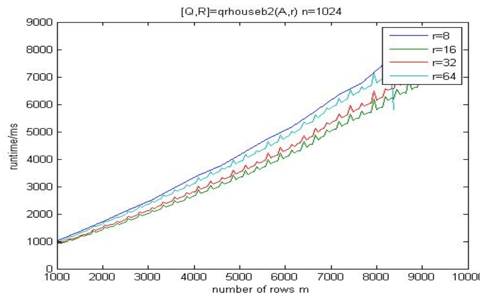
(d) Varying block sizes r with n = 256



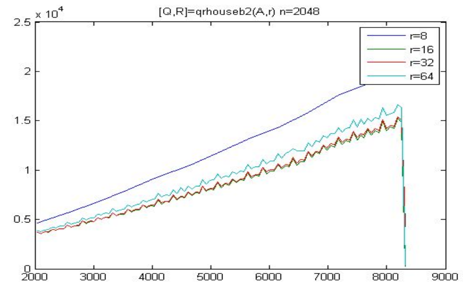
(e) Varying block sizes r with n = 256 with zoom



(f) Varying block sizes r with n = 512

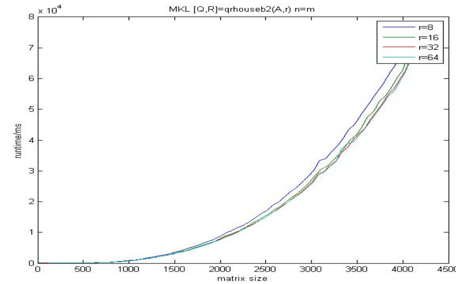


(g) Varying block sizes r with n = 1024

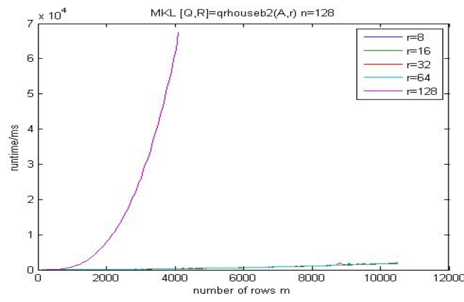


(h) Varying block sizes r with n = 1024

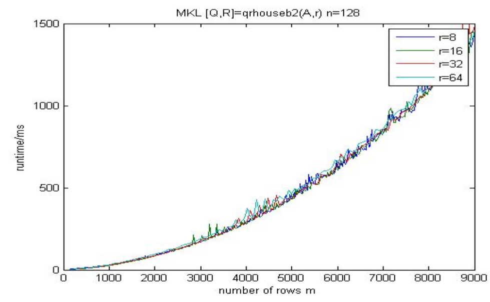
FIGURE 1. Different block sizes on GPU



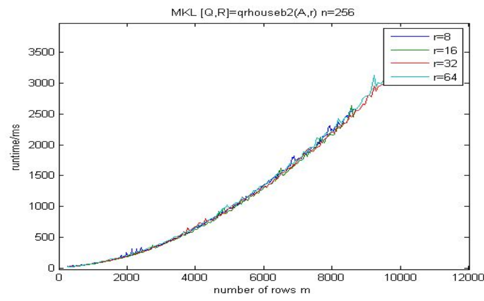
(a) Varying block sizes r with square matrices



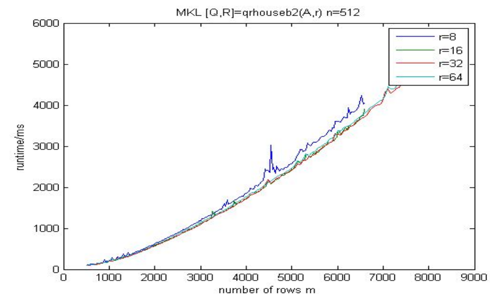
(b) Varying block sizes r with n = 128



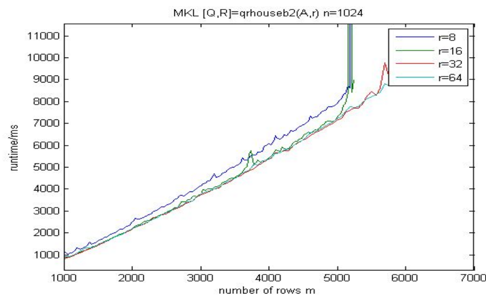
(c) Varying block sizes r with n = 128 with zoom



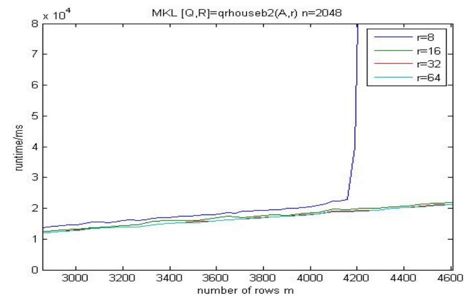
(d) Varying block sizes r with n = 256



(e) Varying block sizes r with n = 512

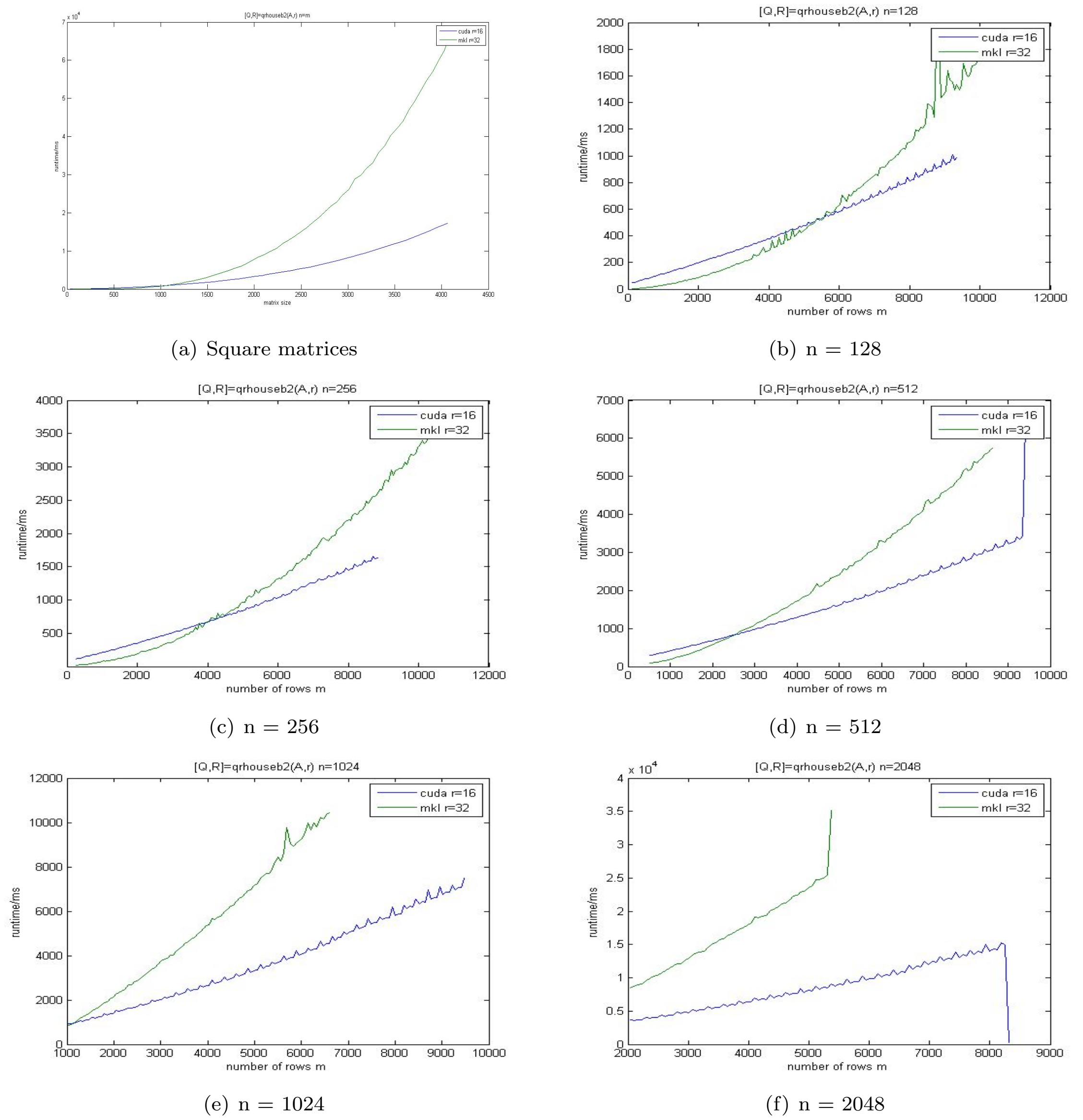


(f) Varying block sizes r with n = 1024



(g) Var. block sizes r with n = 2048

FIGURE 2. Different block sizes on CPU/MKL



(a) Square matrices

(b) n = 128

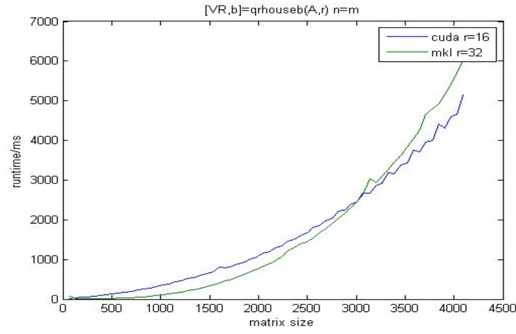
(c) n = 256

(d) n = 512

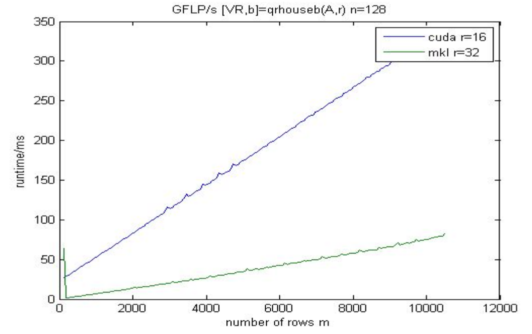
(e) n = 1024

(f) n = 2048

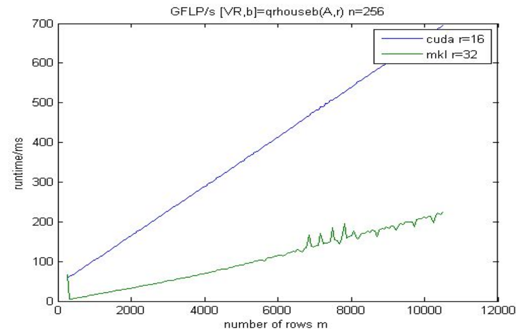
Figure 3. Runtimes QR - GPU vs. CPU



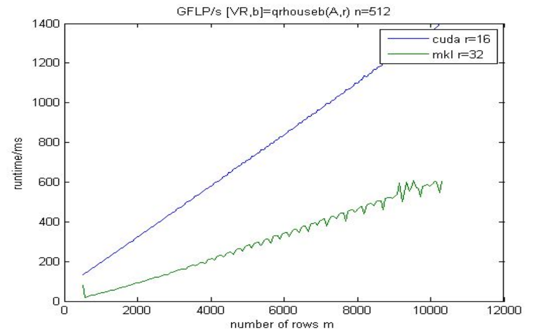
(a) Square matrices



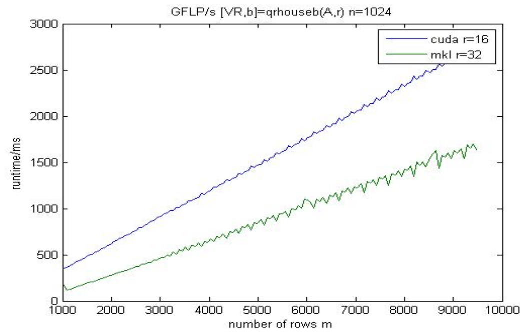
(b) n = 128



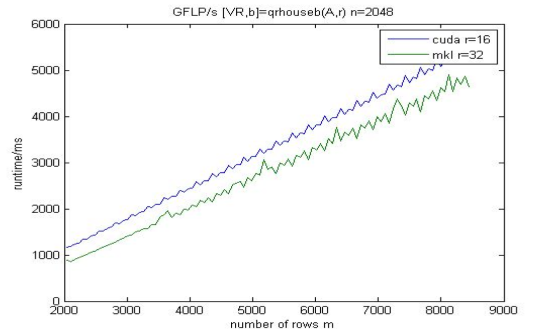
(c) n = 256



(d) n = 512



(e) n = 1024



(f) n = 2048

Figure 4. Runtimes VRβ - GPU vs. CPU

References

[1] Björk, A.: *The calculation of linear least squares problems*, Acta Numerica, Cambridge University Press, 2004, pp. 1-53.
[2] Golub, G.H., van Loan, C.F.: *Matrix Computations*, The John Hopkins University Press, 3rd edition, 1996.
[3] Trefethen, L.N. and Bau III, D.: *Numerical Linear Algebra*, SIAM, 1997.
[4] Mackens, W., Voss, H.:*Mathematik I*, 1st edition, HECO-Verlag, Hamburg, 1993.

(U. Köcher and A. Paprotny) Institute of Computer Technology
Hamburg University of Technology
E-mail address, U. Köcher: uwe.koecher@tuhh.de
URL: http://www.uwe-koecher.de.vu

E-mail address, A. Paprotny: apaprotny@gmx.de