

QR factorization on the GPU

Hallgeir Lien

24th of May

Outline

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

- 1 Introduction to QR
- 2 Computing the QR factorization
- 3 QR factorization on the GPU
- 4 Results

Outline

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

- 1 Introduction to QR
- 2 Computing the QR factorization
- 3 QR factorization on the GPU
- 4 Results

What is QR?

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

The QR factorization of a matrix A gives two new matrices Q and R such that

$$A = QR$$

- Q is orthogonal $\Rightarrow QQ^T = I$
- R is upper triangular

Example

$$\begin{matrix} & A & & Q & & R \\ \begin{bmatrix} 0 & 0 & 6 \\ 1 & 2 & 3 \\ 0 & 4 & 5 \end{bmatrix} & = & \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ -0 & -1 & 0 \end{bmatrix} & \begin{bmatrix} -1 & -2 & -3 \\ 0 & -4 & -5 \\ 0 & 0 & 6 \end{bmatrix} \end{matrix}$$

Applications of QR factorizations

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

QR factorization has a large number of applications

- Finding eigenvalues of matrices
- Calculating the singular value decomposition (SVD)
- The linear least squares problem
- Could also be used for solving linear equations:

$$Ax = \mathbf{b} \Rightarrow QRx = \mathbf{b} \Rightarrow Rx = Q^T \mathbf{b}$$

Application: Eigenvalue finding using QR

GPU QR

Hallgeir Lien

Introduction to QR

Computing the QR factorization

QR factorization on the GPU

Results

We want to find the eigenvalues λ of the matrix A ; i.e. the values λ such that $A\mathbf{v} = \lambda\mathbf{v}$, for some vector $\mathbf{v}$.

- 1 Compute the QR factorization $QR = Q_0 R_0 = A = A_0$.
- 2 Compute $A_1 = R_0 Q_0$
- 3 Compute the QR factorization $Q_1 R_1 = A_1$
- 4 Iterate until convergence.

The real eigenvalues then lie along the diagonal of the last A_k .

Outline

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

- 1 Introduction to QR
- 2 Computing the QR factorization
- 3 QR factorization on the GPU
- 4 Results

Computing the QR factorization

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

Many different ways of computing $A = QR$

- Gram-Schmidt orthogonalization
- Householder reflection transformations
- Givens rotations

I will focus on computing QR by Householder reflector transformations.

Householder reflection transformations

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

Householder reflection transformation matrices H reflects a vector across a hyperplane.

Definition

The householder transformation matrix for real matrices is defined as

$$H = I - 2\mathbf{v}\mathbf{v}^T$$

where $\mathbf{v}$ is the normal of the hyperplane to reflect across.

- Householder reflectors are orthogonal

Computing the QR factorization

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

The goal is to turn the product H^*A into an upper triangular matrix, where H^* is $n - 1$ householder reflectors.

- We choose $\mathbf{v} = \frac{\mathbf{u}}{\|\mathbf{u}\|_2}$
- and $\mathbf{u} = \mathbf{x} + \text{sgn}(x_j)\|\mathbf{x}\|_2\mathbf{e}_1$
where $\mathbf{x}$ is the j 'th column vector of A to be reflected, and $\mathbf{e}_1$ is the unit vector
- $H = I - 2\mathbf{v}\mathbf{v}^T$

Computing the QR factorization

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

$$A = [\mathbf{a}_1 \mathbf{a}_2 \mathbf{a}_3] = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

First step, set $j = 1$ and compute the householder reflector $Q_1 = H_1$. This reflects $\mathbf{a}_1$ to $\mathbf{e}_1$:

$$Q_1 A = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & * & * \end{bmatrix}$$

Computing the QR factorization

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

$$A = [\mathbf{a}_1 \mathbf{a}_2 \mathbf{a}_3] = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

Next, set

$$Q_2 = \begin{bmatrix} I_{1 \times 1} & 0 \\ 0 & H_2 \end{bmatrix}$$

Then,

$$Q_2 Q_1 A = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix} = R$$

Computing the QR factorization

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

We now have

$$A = [\mathbf{a}_1 \mathbf{a}_2 \mathbf{a}_3] = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$Q_2 Q_1 A = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix} = R$$

Setting $Q = Q_1^T Q_2^T$ and multiplying by Q on both sides gives

$$Q^T A = R \Rightarrow QQ^T A = A = QR$$

Outline

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

- 1 Introduction to QR
- 2 Computing the QR factorization
- 3 QR factorization on the GPU
- 4 Results

Blocked Householder QR

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

Calculating QR using the method above is inefficient:

- Lots of rank 1 updates to the identity matrix (from definition of Householder: $H = I - 2\mathbf{v}\mathbf{v}^T$)
- Inefficient memory bandwidth utilization (For GPUs: Lots of global memory accesses).

We need another representation.

We can represent the Householder matrices as

$$Q_{m \times r}^T = Q_1 Q_2 \dots Q_r \dots = I + WY^T$$

where $W_k = [W_{k-1}, Q_{k-1}(2\mathbf{v}_k)]$ and $Y_k = [Y_{k-1}, \mathbf{v}_k]$ ¹

¹C. H. Bischof, C.V. Loan: *The WY Representation for Products of Householder Matrices*

Blocked Householder QR

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

We partition the columns of A into $m \times r$ blocks.

Initialize $Q \leftarrow I$

- ❶ For each block: Compute $Q_{m \times r} = I + WY^T$
 - ❶ Compute r Householder vectors (the $\mathbf{v}$ vectors), and update the current block of A
 - ❷ Compute the W and Y matrices using these vectors.
- ❷ Apply $A_{rest} \leftarrow A_{rest} + YW^T A_{rest}$
- ❸ Apply $Q_{rest} \leftarrow Q_{rest} + YW^T Q_{rest}$

Blocked Householder QR

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

CUBLAS kernels are used as a basis for the GPU implementation.

- 1 Baseline: For-loop is executed as-is on the CPU, calling kernels for each matrix/vector operation.
- 2 Whenever possible, combine multiple CUBLAS calls into a custom kernel
- 3 They utilize shared memory, but they don't say how (except for matrix-vector multiply, on next slide).

Custom matrix-vector multiply

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

A custom matrix-vector multiply kernel $\mathbf{w} = \beta A\mathbf{v} + \alpha\mathbf{w}$ was implemented

- Uses a vector length of 64 (2 warps)
- Keeps $\mathbf{v}$ in shared memory
- One thread computes one element of $\mathbf{w}$; keeps its local result in a register

Speedup over CUBLAS `sgemv`: about $2.5\times$

Outline

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

- 1 Introduction to QR
- 2 Computing the QR factorization
- 3 QR factorization on the GPU
- 4 Results**

Results

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

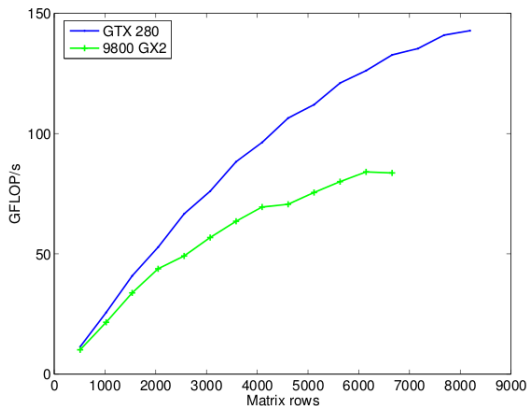


Figure: Sustained GFLOPs rate

Results

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

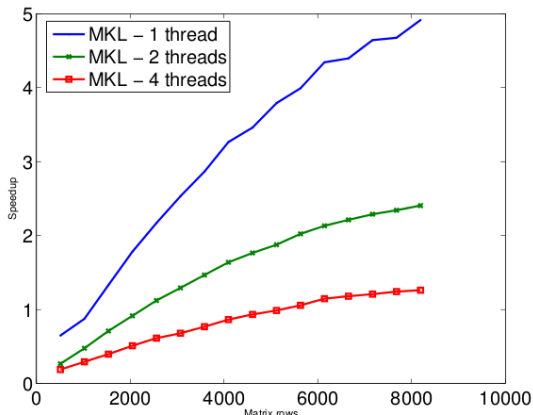


Figure: Speedup over Intel MKL (GTX 280)

Results

GPU QR

Hallgeir Lien

Introduction
to QR

Computing
the QR
factorization

QR
factorization
on the GPU

Results

Results are for single precision only (Realistic?).

Sustained rate is 143 GFLOPS.

The speedup graph doesn't make sense

- CPU used is an Intel Xeon 2.8GHz.
- Speedup over MKL for 4 threads is about 1 for 8000×8000
- Which means that MKL ran at about 143 GFLOPS(!!) for 4 threads
- This is not explained in the paper.