Final project for the Course High Performance Programming, QR Factorization

Philipp Noel von Bachmann

March 20, 2022

1 Introduction

Machine Learning and Data analysis are some of the most influencial fields in our current society. One of the main tasks in Machine Learning is to give a prediction based on some input variables. A common algorithm to use is linear-least-squares, which finds the best fit for a linear model. Here, we need an efficient way to calculate this fit. In Data analysis, a common tool is the Principal Component Analysis, which tries to reduce the dimensionality of the data such that the reconstruction error gets minimized. PCA relies on eigenvalue decomposition. For both methods, we can use the QR decomposition to speed up the computation.

2 Problem description

Suppose $A \in \mathbb{R}^{m \times n}$ is a real, square matrix. Then it can be shown that A can be decomposed into

$$A = QR \tag{1}$$

where Q is orthogonal and R is upper triangular. Finding $Q \in \mathbb{R}^{m \times m}$ and $R \in \mathbb{R}^{m \times n}$ is the task of the algorithm.

3 Solution

We will implement an algorithm known as **Givens Rotation**. This algorithm relies on construction a sequences of matrices G_i, such that when multiplying A with G_i, we get a new matrix with a zero at a predefined place. Choosing G_i such that we eliminate the lower diagonal of A, we get R by multiplying A with all G_i from the left-hand. In addition we get Q by multiplying the identity matrix will all G_i from the right-hand side. We will first show how to eliminate one value at the time by constructing G_i and then how to combine these.

3.1 Prerequisites

3.1.1 Givens rotation

First, we define a Givens rotation matrix as

$$G(i,j,\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c_{ii} & \cdots & -s_{ij} & & \\ & & \vdots & \ddots & \vdots & & \\ & & s_{ji} & \cdots & c_{jj} & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{2}$$

where $c = \cos(\theta)$, $s = \sin(\theta)$ and any not filled out values are 0. Note: Therefore we can equally represent G by $G(i, j, c, s)$.

3.1.2 Eliminating one value

Now the goal is to find a Givens rotation matrix to eliminate/set a specific A_{ij} to 0. This is equivalent to finding θ such that

$$G(i,j,\theta)^T \begin{bmatrix} \times \\ \vdots \\ a \\ \vdots \\ b \\ \vdots \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \vdots \\ r \\ \vdots \\ 0 \\ \vdots \\ \times \end{bmatrix} \tag{3}$$

where $\times$ are arbitrary numbers $a, b \in \mathbb{R}$, $r = \sqrt{a^2 + b^2}$. A trivial solution would be

$$c = \frac{a}{r} \tag{4}$$

$$s = \frac{b}{r} \tag{5}$$

However, r is prone to overflow, so we can instead store it in a different way. If

- $|b| \geq |a|$:
  $t = \frac{a}{b}$, $s = \frac{sgn(b)}{\sqrt{1+t^2}}, c = s \cdot t$
- else:
  $t = \frac{b}{a}$, $c = \frac{sgn(a)}{\sqrt{1+t^2}}, s = c \cdot t$

This leads to the following algorithm

Algorithm 1 computing Givens rotation to eliminate $A_{i,j}$

```
1: procedure GIVENS(A, i, j)
2:     choose k ∈ 0, ... i − 1
3:     compute G(i, k, c, s) with a = A_{k,j}, b = A_{i,j}
4:     return G
```

We note that we can choose $k \in 0, \dots i-1$ arbitrary, the next section will explain this choice in more detail.

3.2 Naive QR-Factorization

3.2.1 Order of elimination

The next question is which order of elimination to choose. Here, we note that if we compute the Givens rotation to eliminate $A_{i,j}$, we then need to multiply GA to eliminate the value. This multiplication affects the whole row i and k of A, where we choose k freely, see algorithm 1. In turn, if we do not want to discard other 0s (say $A_{i,l}$ for example) in those rows again, we need to make sure that $A_{k,l}$ is also 0.

As an more illustrative example, let us assume that we have $R = \begin{bmatrix} x & x & x \\ a & x & x \\ b=0 & c & x \end{bmatrix}$. Now we want to set $c = 0$. This in turn modifies row 1,2 in R afterwards, which means a needs to be 0 in order to keep $b = 0$.

One commonly used scheme is to eliminate all columns after each other from 0 to n-1 and per column go from bottom to top, which looks like this for a 5×5 matrix:

$$\begin{bmatrix} x & x & x & x & x \\ 3 & x & x & x & x \\ 2 & 6 & x & x & x \\ 1 & 5 & 8 & x & x \\ 0 & 4 & 7 & 9 & x \end{bmatrix} \tag{6}$$

where the numbers indicate the order of elimination. It is easy to see that now all previously eliminated columns will stay 0.

3.2.2 Resulting algorithm

The elimination scheme 6 leads to the following algorithm.

Algorithm 2 QR factorization with Givens rotation

```
1: procedure FACTORIZE(A ∈ ℝ^{m×n})
2:     set R = A, Q = I
3:     for j in 1 to n do
4:         for i in m down to j + 1 do
5:             G = GIVENS(A, i, j) with k = i − 1
6:             set R = G(i, j, c, s)R, Q = QG(i, j, c, s)
```

3.3 Serial Optimization

3.3.1 Optimizing Matrix-Multiplication

In general, we could optimize the matrix multiplication with common techniques like cache-blocking, etc. However, if we take a closer look at the Givens rotation matrices, we see that they are very sparse. Additionally, we know in advance which entries of the givens rotation wil be nonzero. Thus we can optimize our matmul. First of all, we can only recalculate columns that will get modified by the multiplication. If we multiply GR, where $G = G(i, j, c, s)$, then we only need to modify rows i, j in R. Secondly, we know

that for calculating those rows we only need to muliply by the c, s in the corresponding row of G, since all other values are 0.

We need further distinguish between multiplication GR and QG. We show the pseudo-code for GR.

Algorithm 3 optimized GR Matmul with Givens

1: **procedure** MATMUL_GR($G = G(i, j, c, s)$, R, *start*)
2: copy the two rows i,j of R into temporary array $T = \begin{bmatrix} R_{i,:} \\ R_{j,:} \end{bmatrix}$
3: **for** k in columns of R larger than *start* **do**
4: $R_{i,k} = c \cdot T_{1,k} - s \cdot T_{2,k}$
5: $R_{j,k} = s \cdot T_{1,k} + c \cdot T_{2,k}$
return R

We can also start only changing values from column *start* on, since from our general algorithm 2 we know that all values in columns smaller than k are already eliminated. Note that this is only true for the multiplication GR, not QG.

This optimization is optimal from a serial perspective, since each of the elements in the respective rows and columns we recompute values in could potientially get modified by our rotation. Therefore, we cannot reduce the matmul to fewer operations.

3.3.2 Memory optimization

The improved matrix multiplication algorithm 3 shows that we only need to access two rows per multiplication to perform the update of R, for Q we need to access two columns however. Ultimately, we want to access matrix elements as close to each other as possible, since by reading one element, we read the whole cache-line and have faster access to nearby elements in the following steps. Therefore, it makes sense to store R row-wise, and Q column-wise. In that way, we get the most optimal data locality for Q and R.

3.3.3 Further serial optimizations

The following further small optimizations were thought, although some were rejected for the following reasons:

- Wrote pow calls out (however compiler flag probably does it already) (accepted)
- Inlined functions by using the compiler hint **inline** (accepted)
- Declared the size of A as constant to help compiler do some optimizations (accepted)
- Do not unroll loops, since there are no loops whose size is certain at compile times (rejected)

3.3.4 Using optimizer flags

Of course we tried all common optimizer flags, to further optimize the code, see result 5.6 for discussion.

3.4 Parallel optimization

In order to parallelize our code, we first need to check which parts can be done in parallel. First, note that the computation of the Givens rotation in line 5 of algorithm 2 is a pure function, eg. just depends on the inputs and does not have any side effects. Thus, we only need to focus on updating Q, R in line 6. Here, it is necessary to distinguish in two cases:

- For left-hand Givens rotation GR, if $G = G(i, j, c, s)$, only rows i, j of R get modified
- For right-hand Givens rotation QG, if $G = G(i, j, c, s)$, only columns i, j of Q get modified

Consequently, two update steps rotating around $G(i_1, j_1, c_1, s_1)$ and $G(i_2, j_2, c_2, s_2)$ are independent, if i_1, i_2, j_1, j_2 are pairwise distinct. This means we need to make sure that if we set $R_{i,j}$ to 0 in one iteration, we can set any other elements in Row i, j to 0 in the same iteration, in order to not interfere between the computations.

In combination with the constraint that if we want to set $R_{i,j}$ to 0, $R_{i,:j}$ and $R_{i-1,:j}$ already need to be 0, (see Section 3.2.1) we arrive at the following scheme, shown for a 5×5 matrix:

$$\begin{bmatrix} x & x & x & x & x \\ 3 & x & x & x & x \\ 2 & 4 & x & x & x \\ 1 & 3 & 5 & x & x \\ 0 & 2 & 4 & 6 & x \end{bmatrix} \tag{7}$$

where the values denote in which iteration we can set the element to 0 without cross-interference. This shows that we can set $R_{2,0}$ and $R_{4,1}$ to 0 in parallel for example.

This scheme is optimal since we can only eliminate $R_{i,j}$ if $R_{i,:j}$ and $R_{i-1,:j}$ are already 0. This upper-bounds the number of rows we can update in each iteration. Furthermore, in each of those iterations we update all possible rows, so there is no way of updating more elements without cross-interfering.

We arrive at the parallel algorithm:

Algorithm 4 parallel QR factorization

```
1: procedure FACTORIZE_PARALLEL(A ∈ ℝ^{m×n})
2:     set R = A, Q = I
3:     for it in 1 to (n − 1) ∗ 2 + m − n do
4:         In parallel:
5:         for i, j given by the iteration scheme 7 do
6:             G = GIVENS(A, i, j) with k = i − 1
7:             set R = G(i, j, c, s)R, Q = QG(i, j, c, s)
```

The code was getting parallelized with **OpenMP**. Since we have equal load in line 6+7 of algorithm 4, we do not need to use load balancing.

Looking at 7, we cannot expect a linear scaling with the number of threads. For example in iteration 0 and 1, we only have 1 element so there is no speedup by using more threads. More formally if $A \in \mathbb{R}^{m \times n}$, then we can have a maximum of $\min(m/2, n)$ computations running in parallel, achieved in the iteration where we set $R_{1,0}$ to 0. In general, we get more elements to eliminate in parallel the further the iterations.

4 Code

The code is structured the same way as the solution described. We include 3 different versions:

- **qr_base.c**: The baseline algorithm 1.
- **qr_serial.c**: The serial optimized algorithm, this means the whole section 3.3
- **qr.c**: The final algorithm, with all optimization.

In the file **qr.c** the code is commented, the others files have mostly the same functions so we did not comment these. For building the code, a **Makefile** is included.

5 Experiments and Results

5.1 Hardware information

- CPU: Apple M1 with 4 performance cores and 4 efficiency cores
- GPU: Apple M1 8 core.
- Compiler: Slang, Version 13.0.0
- Plugged into power adapter, this makes a big difference!

5.2 Correctness of code

In order to evaluate the correctness of the code, we run a test script. First, we check that R is upper triangular, here we allow a small tolerance, since the computation of the Givens rotation even in its more stable form 3.1.2 has some rounding errors. However we set the lower diagonal afterwards to 0, in order to ensure correct results. Then we verify that mutiplying QR indeed results in A.

Note that we did not optimize this part of the code, since in practice the verification is not required to get to the solution.

5.3 Format of Results

In the following section, we will show timing results for different matrix sizes. We denote the sizes by m, n, which we defined as decomposing the matrix $A \in \mathbb{R}^{m \times n}$, therefore $Q \in \mathbb{R}^{m \times m}$ and $R \in \mathbb{R}^{m \times n}$.

5.4 Baseline Results

The following table shows the results for the base algorithm 2.

m \ n	10	25	50	100
10	0.0005	0.0008	0.001	0.0018
25	0.018	0.034	0.049	0.082
50	0.235	0.611	1.076	1.602
100	3.623	9.512	19.525	34.626

By varying n, the runtime increases approximately linearly. We can distinguish between two cases:

- $n \geq m$ Increasing n will not introduce more values to be eliminated, but increase the cost of the matrix multiplication GR, see algorithm 2, line 6.
- $n < m$ Here, increasing n will introduce new value to get eliminated in addition to more costly matrix multiplications.

By varying m, the runtime increases superlinear, because we need to eliminate more values and the matrix multiplications become more expensive. Again we can distinguish:

- $m \geq n$ Increasing m by x will introduce exactly $n \cdot x$ new values to eliminate.
- $m < n$ Increasing m by x will introduce $m + 1 + m + 2 + m + 3 + \cdots + m + x$ new values to eliminate.

5.5 Optimizing the Matrix-Multiplication

Optimizing matmul with algorithm 3 leads to the following results:

m \ n	10	25	50
50	0.0003	0.0007	0.0011
500	0.019	0.040	0.079

We see that for $m = 50, n = 50$ we get a runtime of roughly $\frac{1}{1000}$ of the original runtime. Therefore, optimizing matrix-multiplication is very effective. Where before we needed to perform $m \cdot m \cdot n$ additions and multiplications to perform QR, now we need $4 \cdot n - start$ operations, where *start* is our current columns, see algorithm 3.

5.6 Compiler flags

For $m, n = 500$:

Baseline	-O1	-O2	-O3	-03 -march=native -ffast-math
0.747	0.167	0.137	0.134	0.119

Consequently, we choose **-03 -march=native -ffast-math** as optimization flag from now on.

5.7 Memory optimization

We show the effect of storing Q column wise for both modifying m, n. The baseline denotes the naive storing of both Q, R row-wise.

For $n = 1000$:

m	100	500	1000
baseline	0.0061	0.147	0.999
optimized	0.0047	0.115	0.590

For $m = 1000$:

n	100	500	1000
baseline	0.135	0.612	0.991
optimized	0.067	0.339	0.609

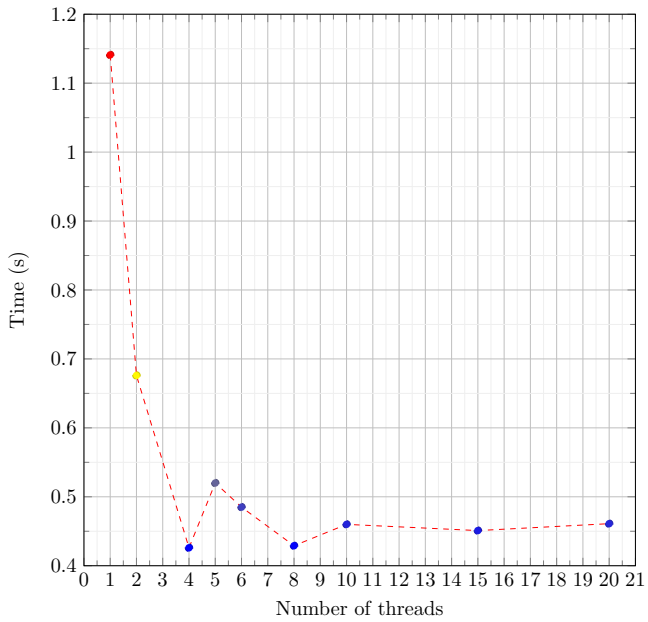
We see that by varying m, n, we get a consistently better runtime for the column-wise storage. This is because already for small m, n, the multiple rows will not fit in one

cacheline and therefore we need to jump often between columns to update them. For a column-wise storage however, we pass through the whole cacheline one by one and therefore do not need to get new data from memory that often.

5.8 Parallelization

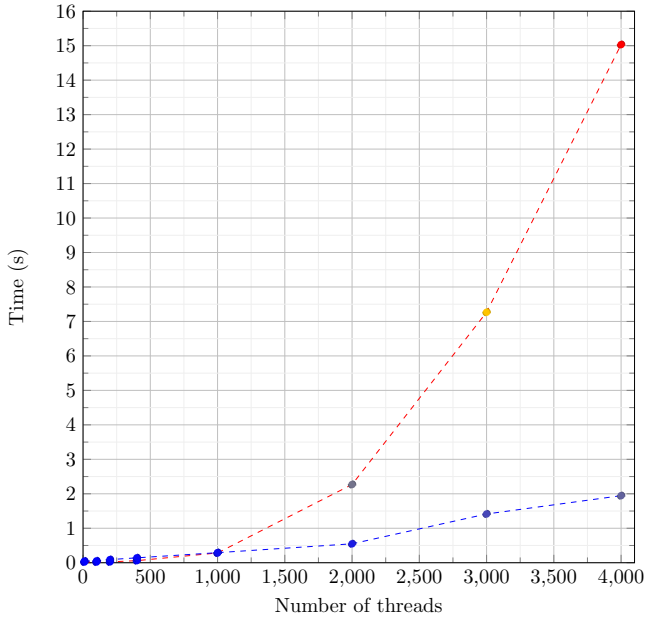
Because the computation of m, n is a bit more involved now, see 7, we get a slightly higher baseline with 1.137s compared to 0.986s for our serial code and $m, n = 1000$.

However when increasing the number of threads, and keeping $m, n = 1000$:



The plot shows that we only get a sublinear decrease in runtime by increasing the number of threads. As already discussed in 3.4, this is because we have some data-dependencies that prohibit more parallelization. Interestingly, we get an increase for 5 threads compared to 4. This is probably because the machine for testing has 4 high-performance and 4 efficiency cores, so by using 5 threads the high-performance cores have to wait for the efficiency ones.

As also already discussed we get more parallelization the further our iterations are. Therefore, we also change the matrix sizes (m in red, n in blue) for a constant size of threads (4 since this was the best value in the previous result):



When comparing to the results 5.7, we see that from $m = 100$ to $m = 1000$, we now get an increase of approximately 20 times, versus 120 times before. This means that while we still have an superlinear increase in total, it is less than before. In the case of n we see that we have a slightly higher than linear increase, which is on par with earlier results 5.4.

6 Conclusion

This report shows how to perform several optimization to reduce the runtime of the QR Factorization. First, we performed some serial optimizations, most importantly a new matrix-multiplication which made use of a sparse Givens Matrix G. This turned out to be very effective in reducing the runtime. Secondly, we showed a way to parallelize the code. Here, we can eliminate multiple elements in parallel. This however requires an elaborate scheme of iterating over the elements, and does not get linear increase with the number of threads since we have multiple data dependencies. The final algorithm was able to perform the task significantly faster than the naive implementation.

7 References

- `https://www.math.usm.edu/lambers/mat610/sum10/lecture9.pdf`
- `https://www.math.usm.edu/lambers/mat610/class0208.pdf`
- `https://en.wikipedia.org/wiki/Givens_rotation`

Note: Even if these pages contain code, no code was used from these pages. Only the explanations were used to get an understanding of the algorithm, all sections and the code are written independently.