A Massively Parallel Eigenvalue Solver for Small Matrices on Multicore and Manycore Architectures

Manfred Liebmann
Technische Universität München
Chair of Optimal Control
Center for Mathematical Sciences, M17
manfred.liebmann@tum.de

January 18, 2016

Motivation

Optimization problem in magnetic resonance imaging (MRI) using nuclear magnetic resonance (NMR) capabilities of the magnetic resonance scanner for fat / water quantification. Analyzing the magnitude and phase images of multiple echoes allows the computation of the fat fraction of tissue. (K. Bredies)

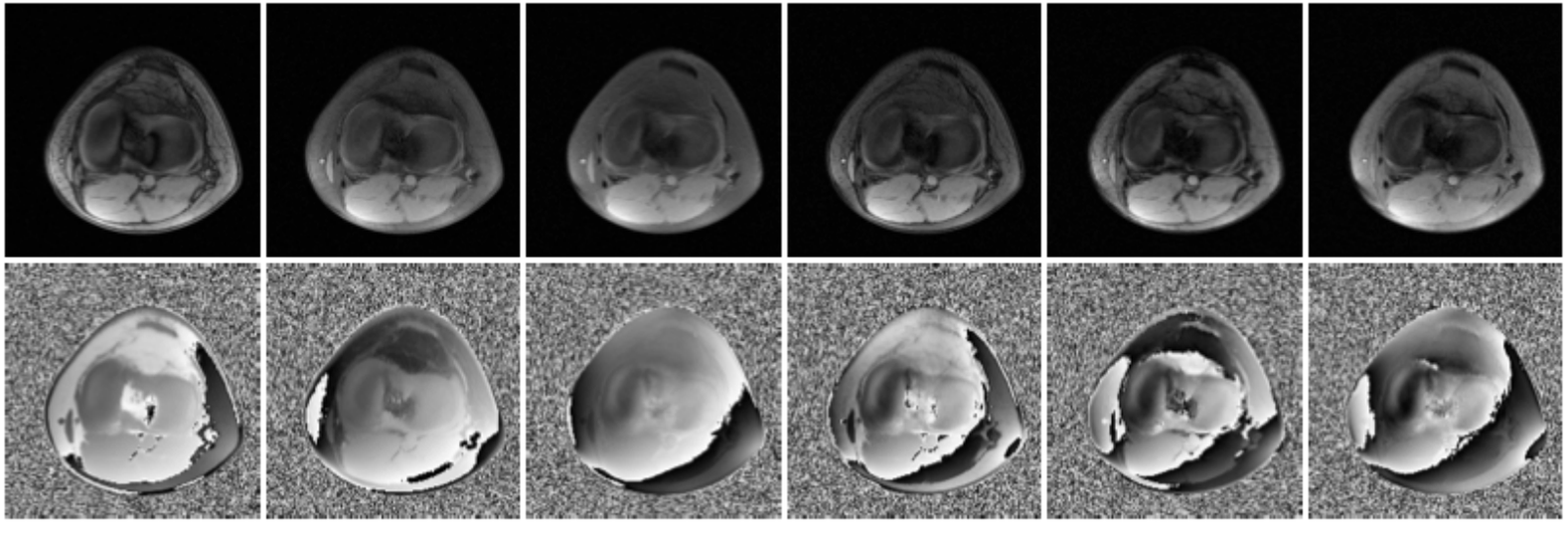

Figure 1: Magnitude and phase images form multiple echoes

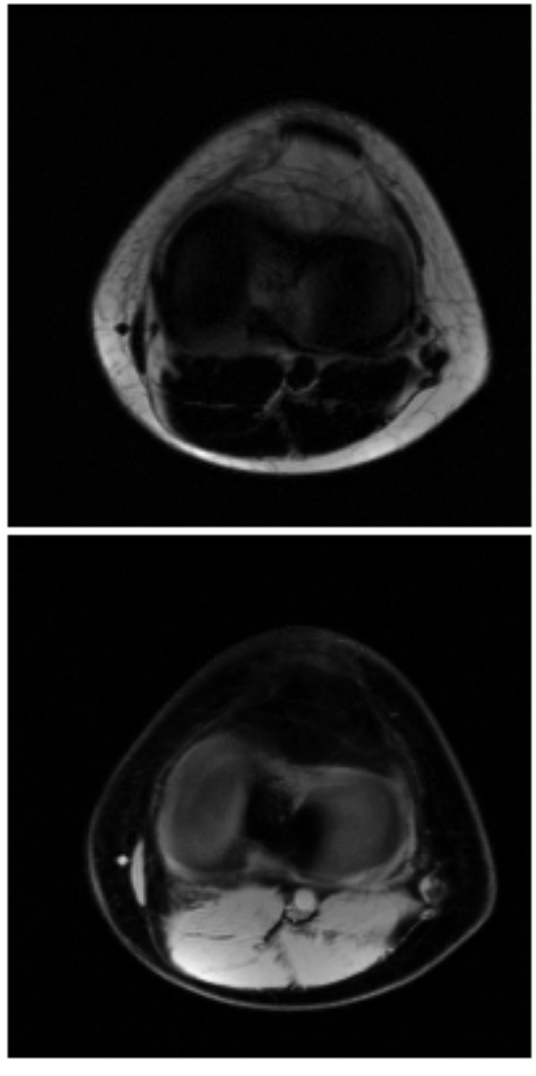

Figure 2: Calculated fat and water separation in tissue

Optimization Problem

Assume $t_m = t_0 + m\delta t$ are equispaced echo times and R_2^* the relaxation parameter inhomogeneity image given, then the least squares residual

$$r^2(\delta B_0) = \min_{\alpha_w, \alpha_f} \sum_{m=1}^{M} |s_m - (\alpha_w w_m + \alpha_f f_m) e^{2\pi i \gamma \delta B_0 t_m} e^{R_2^* t_m}|^2 \tag{1}$$

is a *trigonometric polynomial* using the substitution $z = e^{2\pi i \gamma \delta B_0 \delta t}$ and the local minima of the functional are determined by the roots of the polynomial. Where α_w, α_f are the water and fat intensity images, δB_0 is the field inhomogeneity image and w_m and f_m are the ideal water and fat signals.

Find the roots of a polynomial for every pixel!

Companion Matrix

Let $p(z) = a_0 + a_1 z + \cdots + a_{n-1} z^{n-1} + z^n$ be a complex polynomial.

The *companion matrix* of the polynomial,

$$A = \begin{pmatrix} 0 & & & & -a_0 \\ 1 & 0 & & & -a_1 \\ & 1 & \ddots & & -a_2 \\ & & \ddots & 0 & \vdots \\ & & & 1 & -a_{n-1} \end{pmatrix} \in \mathbb{C}^{n \times n}, \tag{2}$$

has the characteristic polynomial $\chi_A(z) := \det(zE - A) = p(z)$. Companion matrices are Hessenberg matrices!

The `roots` command in Matlab uses this approach to calculate the roots of a polynomial.

The Long History of LAPACK

LAPACK was first written in the 70s / 80s with a different computer architecture in mind than we have today.

- **Good software: 70s / 80s**
  - Sequential algorithms
  - FLOPS are the main concern!
  - Memory access patterns are not so important

- **Good software: Today**
  - Massively parallel hardware architectures need efficient parallel algorithms
  - Efficient memory access is extremely important!
  - Parallelization usually for large scale problems
  - Expose parallelism for many small scale problems through **vectorization**!

Vectorization of the QR Algorithm ZLAHQR

It is very hard to derive any meaningful parallelism within a single small eigenvalue problem.

However it is still possible to *vectorize* the algorithm and thus solve many small eigenvalue problems simultaneously.

The LAPACK implementation of the core QR algorithm ZLAHQR has complex program flow.

Can vectorized code of a complex program flow be still efficient?

Lets have a look at the algorithm!

Algorithm 1: ZLAHQR Body

```
for I = IHI to ILO step −1 do
    L = ILO;
    for ITS = 0 to 30 do
        Deflation;
        if L ≥ I then
            break;
        I1 = L;
        I2 = I;
        if ITS = 10 then
            Exceptional Shift A;
        else if ITS = 20 then
            Exceptional Shift B;
        else
            Wilkinson Shift;
        QR Single Shift;
for I = IHI to ILO, step −1 do
    E[I] = H[I, I];
```

Algorithm 2: Deflation

for $K = I$ *to* $L + 1$, *step* -1 **do**
  $w_4 = H[K, K-1]$;
  if $|w_4| \leq smlnum$ **then**
    break;
  $w_1 = H[K-1, K-1]$, $w_2 = H[K, K]$, $tst = |w_1| + |w_2|$;
  if $tst = 0$ **then**
    if $K - 2 >= ILO$ **then**
      $w_3 = H[K-1, K-2]$, $tst = tst + |w_3|$;
    if $K + 1 <= IHI$ **then**
      $w_6 = H[K+1, K]$, $tst = tst + |w_6|$;
  if $|w_4| \leq ulp \cdot tst$ **then**
    $w_5 = H[K-1, K]$;
    $ab = \max(|w_4|, |w_5|)$, $ba = \min(|w_4|, |w_5|)$;
    $aa = \max(|w_2|, |w_1 - w_2|)$, $bb = \min(|w_2|, |w_1 - w_2|)$;
    $s = aa + ab$;
    if $ba \cdot (ab/s) \leq \max(smlnum, ulp \cdot (bb \cdot (aa/s)))$ **then**
      break;
$L = K$;
if $L > ILO$ **then**
  $H[L, L-1] = 0$;

Algorithm 3: Exceptional Shift A

$t = |\Re(H[L+1, L])| \cdot 0.75 + H[L, L]$;

Algorithm 4: Exceptional Shift B

$t = |\Re(H[I, I-1])| \cdot 0.75 + H[I, I]$;

Algorithm 5: Wilkinson Shift

$t = H[I, I]$;
$h_2 = H[I-1, I]$;
$h_3 = H[I, I-1]$;
$u = \sqrt{h_2} \cdot \sqrt{h_3}$;
$s = |u|$;
if $s \neq 0$ **then**
    $h_4 = H[I-1, I-1]$;
    $x = (h_4 - t) \cdot 0.5$;
    $s = \max(s, |x|)$;
    $z_1 = x/s$, $z_2 = u/s$;
    $y = \sqrt{z_1^2 + z_2^2} \cdot s$;
    if $|x| > 0$ **then**
        $z = x/|x|$;
        if $\Re(z) \cdot \Re(y) + \Im(z) \cdot \Im(y) < 0$ **then**
            $y = -y$;
    $v = u$;
    $t = t - u \cdot (u/(x+y))$;

Algorithm 6: QR Single Shift

for $K = M$ *to* $I - 1$ **do**
  if $K > M$ **then**
    $v_0 = H[K, K-1]$;
    $v_1 = H[K, K]$;
  if $v_1 = 0$ *and* $\Im(v_0) = 0$ **then**
    $t_1 = 0$;
  Householder reflector;
  if $K > M$ **then**
    $H[K, K-1] = v_0$;
    $H[K, K] = 0$;
  $t_2 = \Re(t_1) \cdot \Re(v_1) - \Im(t_1) \cdot \Im(v_1)$;
  Householder reflector from the left;
  Householder reflector from the right;
  $z = H[I, I-1]$;
  if $\Im(z) \neq 0$ **then**
    $d = |z|$;
    $z = z/d$;
    $\Im(z) = -\Im(z)$;
    $H[I, I-1] = d$;
    Scale rows with z and columns with $\bar{z}$;

Algorithm 7: Householder reflector

$norm = \sqrt{\Re(v_0)^2 + \Im(v_0)^2 + \Re(v_1)^2 + \Im(v_1)^2}$;
$beta = \mathrm{copysign}(norm, \Re(v_0))$;
$\Re(z) = \Re(v_0) + beta$;
$\Im(z) = \Im(v_0)$;
$\Re(t) = \Re(z)$;
$\Im(t) = \Im(z)$;
$z = 1/z$;
$v_1 = v_1 \cdot z$;
$t = t/beta$;
$\Re(v_0) = -beta$;
$\Im(v_0) = 0$;

Algorithm 8: Householder reflector from the left

$i_1 = (K + K \cdot n) \cdot m - off;$
$i_2 = ((K + 1) + K \cdot n) \cdot m - off;$
for $j = K$ *to* I_2 **do**
  $h_1 = H[i_1];$
  $h_2 = H[I_2];$
  $z = h_1 \cdot \bar{t}_1 + h_2 \cdot t_2;$
  $H[i_1] = H[i_1] - z;$
  $H[i_2] = H[i_2] - z \cdot v_1;$
  $i_1 = i_1 + n \cdot m;$
  $i_2 = i_2 + n \cdot m;$

Algorithm 9: Householder reflector from the right

$i_1 = (I_1 + K \cdot n) \cdot m - off;$
$i_2 = (I_1 + (K + 1) \cdot n) \cdot m - off;$
for $j = I_1$ *to* $\min(K + 1, I)$ **do**
  $h_1 = H[i_1];$
  $h_2 = H[I_2];$
  $z = h_1 \cdot t_1 + h_2 \cdot t_2;$
  $H[i_1] = H[i_1] - z;$
  $H[i_2] = H[i_2] - z \cdot \bar{v}_1;$
  $i_1 = i_1 + m;$
  $i_2 = i_2 + m;$

Array of Structures (AoS) or Structure of Arrays (SoA) or Hybrid

Vectorization requires a uniform coalesced memory access for optimal performance

- **Which data layout is preferable?**
  - Array of Structures (AoS) layout is most of the time **not suitable** for vectorization!
  - Structure of Arrays (SoA) is the natural data layout for vectorization
  - SoA problem: Memory access to far apart addresses leads to performance penalties
  - Optimal: **Hybrid**: Many small SoA blocks adapted to the hardware vector length
  - CPU: $8 - 16$ matrices in a SoA block. L1 cache friendly layout.
  - GPU: $32 - 256$ matrices in a SoA block.

The vector length is a natural tuning parameter that can be adapted for various hardware.

Hybrid Data Layout for Vectorization

Let $k \in \mathbb{N}$ and let $A^{(n)} \in \mathbb{C}^{k \times k}$ and $1 \leq n \leq N$ and let $M = 2^m$ be the vector length. Then we get N/M interleaved matrix blocks.

$$\mathbb{A}^{(1)} = \begin{bmatrix} a_{1,1}^{(1)} & a_{1,1}^{(2)} & \dots & a_{1,1}^{(M)} \\ \vdots & \vdots & & \vdots \\ a_{1,k}^{(1)} & a_{1,k}^{(2)} & \dots & a_{1,k}^{(M)} \\ \vdots & \vdots & & \vdots \\ a_{k,1}^{(1)} & a_{k,1}^{(2)} & \dots & a_{k,1}^{(M)} \\ \vdots & \vdots & & \vdots \\ a_{k,k}^{(1)} & a_{k,k}^{(2)} & \dots & a_{k,k}^{(M)} \end{bmatrix} \tag{3}$$

Continue with the remaining $L := N/M$ blocks $\mathbb{A}^{(l)}$, $1 \leq l \leq L$ and we get the full hybrid data layout: $\mathbb{A}^{(1)}, \dots, \mathbb{A}^{(L)}$.

Performance Results for Different Libraries and New Code

A test image with 384×384 pixels gives 147456 eigenvalue problems. A benchmark run below calculates the eigenvalues of 3 different images with a total of 442368 matrices of dimension 10×10.

Library	Routine	Time
ATLAS	zgeev	14.4191s
ATLAS	zlahqr	14.1347s
MKL	zgeev	6.12243s
MKL	zlahqr	2.06272s
NEW	(compatible)	1.79652s
NEW	(tuned)	1.16941s
NEW	(deflation)	**1.04749s**

Table 1: Comparison of different LAPACK libraries and the new code on a compute node with 2x Intel Xeon E5-2650 @ 2.0GHz (16 cores) using OpenMP parallelization

Performance Results for Accelerators

Hardware	Cores	Compiler	Time
2x Intel Xeon E5-2650	16	g++ openmp	1.04749s
1x Intel Xeon Phi 5110P	60	icpc offload	4.93889s
1x Intel Xeon Phi 5110P	60	icpc native	2.14887s
1x Nvidia GTX 480	448	pgcpp openacc	0.747283s
1x Nvidia Tesla K20	2496	pgcpp openacc	1.07871s
1x Nvidia GTX 480	448	nvcc sm20	0.58854s
1x Nvidia Tesla K20	2496	nvcc sm35	**0.48903s**

Table 2: Timings for calculating all eigenvalues of 442368 companion matrices

All programs are compiled form the same source code!

The only specific modifications are pragmas decorating the outer loop and CUDA thread ids replacing the outer loop.

Conclusions

- **Hardware has changed!**
  - Old code isn't efficient any more on modern architectures
  - Hardware characteristics have significantly changed over the years
  - FLOPS are free and memory access is expensive
  - Memory access patterns are very important for efficient code
  - Multicore CPUs and Manycore GPUs evolve towards a common architecture
  - **Single Instruction Multiple Thread (SIMT)** is a good model for vectorization (GPUs)
  - SIMT emulation for CPUs with AVX vector intrinsics already possible
  - *Vectorization* is key to efficient code
  - **Future work:** Explicit vectorization library using the SIMT model for CPUs