Accelerated and Validated Model Based MR Signal Reconstruction

Drew Mitchell
Summer 2014 Tutorial Write-Up
Faculty Member: R. Jason Stafford, Ph.D.

August 19, 2014

Abstract

MR signal reconstruction can be accelerated by parallelizing as many tasks as possible during the reconstruction. The purpose of this tutorial was to use the CUDA parallel computing platform to create a kernel which parallelizes one of the reconstruction tasks: polynomial root solving. The Gram-Schmidt QR method was selected to accomplish this task due to its relative simplicity. The more complex Francis double-shift algorithm was investigated as an alternative. No numerical eigenvalue method guarantees convergence for every matrix, but the Francis algorithm converges for a larger set of problems. The kernel using the Gram-Schmidt method was integrated into the rest of the reconstruction algorithm. If the Gram-Schmidt method is insufficient for these purposes, the Francis algorithm kernel will be completed and used instead. Wavelet and curvelet transforms were also investigated for their use in reconstruction of sparse signals.

Objectives

The objective for this tutorial was to write kernels to perform linear algebra techniques, such as Gaussian elimination and QR decomposition, on the GPU in order to accelerate an MR signal reconstruction algorithm as it calls these subroutines. My personal objectives were to become proficient with C++, MATLAB, the CUDA parallel computing platform, and computational methods and algorithms commonly employed in MR signal reconstruction.

Background

GPGPU Computing

General purpose graphics processing unit (GPGPU) computation is an ideal way to accelerate reconstruction, because many of the subroutines of reconstruction involve parallelizable tasks which may be run simultaneously rather than serially. Traditional computation is performed with a central processing unit (CPU), which typically accomplishes one task per processor core in a linear manner. Most modern CPUs possess multiple cores, so some limited parallel processing is available. GPUs, on the other hand, contain many more cores capable of running hundreds of threads in parallel. In recent years, this parallel processing power has been harnessed for computation in many scientific fields because of its ability to accelerate common programming tasks by as much as a hundredfold.

For this application, a set of data is associated with each pixel of the image to be reconstructed. Calculations involving the data of one pixel may be performed independently of those for other pixels. Problems of this type are known as embarrassingly parallel. The computation is accelerated by performing calculations simultaneously on as many of the 256×256 pixels as possible.

Reconstruction Algorithm

This reconstruction process requires a quick and parallel method for solving for the roots of a polynomial. The Steiglitz-McBride algorithm is applied to the signal and returns the coefficients of the polynomial for

each pixel. The numerical method used to solve for roots of a polynomial gernerally involves converting the polynomial into its companion matrix and solving for the eigenvalues. Two techniques for determining the eigenvalues of a matrix are described below, the Gram-Schmidt QR method and the Francis Double-Shift algorithm. All eigenvalue methods are iterative and converge to a solution after a sufficient number of iterations. Afterward, image values are computed from the eigenvalues, which are the roots of the polynomial.

Methods/Techniques

Root Solving by QR Decomposition

To find the roots of a polynomial the Companion matrix method can be used. It can be verified by direct computation that the so called Companion matrix

$$\begin{pmatrix} 0 & 0 & \cdots & 0 & -c_0 \\ 1 & 0 & \cdots & 0 & -c_1 \\ 0 & 1 & \cdots & 0 & -c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 1 & -c_{n-1} \end{pmatrix}$$

has the characteristic polynomial $p(t) = c_0 + c_1 t + \cdots + c_{n-1}t^{n-1} + t^n$. Thus the eigenvalues of the companion matrix are the roots of $p(t)$. To find the eigenvalues the QR algorithm can be used. The algorithm performs iterations of the form

$$A_{k+1} = R_k Q_k,$$

where Q_k is an orthogonal matrix and R_k an upper triangular matrix, such that $A_k = Q_k R_k$, i.e. the QR decomposition of A_k, and $A_0 = A$. It can be shown, that A_k has the same eigenvalues as A and that it converges to a triangular matrix, the *Schur form*. Thus, the eigenvalues of A can be read off the diagonal of A_k after convergence.

Gram-Schmidt Orthogonalization

One method for performing QR decomposition is Gram-Schmidt orthonormalization. This is a process for orthonormalizing a set of vectors in an inner product space, and the algorithm facilitates QR decomposition when applied to the column vectors of a matrix. If projection is abbreviated such that $\text{proj}_u(v) = \frac{\langle u,v \rangle}{\langle u,u \rangle} u$, then the set of vectors v are transformed to the set of orthogonal vectors u by the following process:

$$\begin{aligned} u_1 &= v_1 \\ u_2 &= v_2 - \text{proj}_{u_1}(v_2) \\ u_3 &= v_3 - \text{proj}_{u_1}(v_3) - \text{proj}_{u_2}(v_3) \\ &\vdots \\ u_k &= v_k - \textstyle\sum_{j=1}^{k-1} \text{proj}_{u_j}(v_k) \end{aligned}$$

The orthogonal vectors u are normalized to the set of unit vectors $e_k = \frac{u_k}{\|u_k\|}$. When applying Gram-Schmidt orthonormalization to QR decomposition, the set of column vectors in A, such that $A = [a_1, \ldots, a_n]$, are orthogonalized.

$$\begin{aligned} u_1 &= a_1 \\ u_2 &= a_2 - \text{proj}_{e_1}(a_2) \\ u_3 &= a_3 - \text{proj}_{e_1}(a_3) - \text{proj}_{e_2}(a_3) \\ &\vdots \\ u_k &= a_k - \textstyle\sum_{j=1}^{k-1} \text{proj}_{e_j}(a_k) \end{aligned}$$

Once the set of vectors $[a_1, \ldots, a_n] = A = QR$ have been orthonormalized, the orthogonal matrix Q and upper triangular matrix R are reconstructed as follows:

$$Q = [e_1, \ldots, e_n]$$

$$R = \begin{pmatrix} \langle e_1, a_1 \rangle & \langle e_1, a_1 \rangle & \langle e_1, a_3 \rangle & \cdots \\ 0 & \langle e_2, a_2 \rangle & \langle e_2, a_3 \rangle & \cdots \\ 0 & 0 & \langle e_3, a_3 \rangle & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

The classical Gram-Schmidt process is numerically unstable. A modified Gram-Schmidt algorithm corrects this instability by also orthogonalizing $u_k^{(i)}$ against rounding errors in $u_k^{(i-1)}$:

$$\begin{aligned} u_k^{(1)} &= v_k - \text{proj}_{u_1}(v_k) \\ u_k^{(2)} &= u_k^{(1)} - \text{proj}_{u_2}(u_k^{(1)}) \\ &\vdots \\ u_k^{(k-2)} &= u_k^{(k-3)} - \text{proj}_{u_2}(u_k^{(k-3)}) \\ u_k^{(k-1)} &= u_k^{(k-2)} - \text{proj}_{u_2}(u_k^{(k-2)}) \end{aligned}$$

The GPU kernels were written in C++ with calls to functions from the CUDA toolkit. The kernel itself was driven by a main program written in MATLAB for debugging before integration into the reconstruction algorithm. The efficiency of parallel structure of the kernel was analyzed using Nvidia Visual Profiler, an application included with the CUDA toolkit to aid in the efficient parallelization of code. The Gram-Schmidt orthogonalization and root solving kernel and the MATLAB driver program are provided in Appendix A.

The Francis Double-Shift Algorithm

The Francis Double-Shift algorithm requires first that the matrix be transformed to its upper Hessenberg form via Householder reflections. A significant limitation of this requirement is that it usually generates complete fill-in of sparse matrices. The Francis algorithm uses shifts along the matrix diagonal–denoted by σ–to accelerate the convergence of the matrix to *Schur form.*

$$\begin{aligned} A_0 - \sigma_1 I &= Q_1 R_1 \\ A_1 &= R_1 Q_1 + \sigma_1 I \\ A_1 - \sigma_2 I &= Q_2 R_2 \\ A_2 &= R_2 Q_2 + \sigma_2 I \end{aligned}$$

The value of the shifts are adjusted each iteration so that σ is close to one of the eigenvalues. The use of two shifts in each iteration allows the Francis algorithm to solve for complex conjugate pair eigenvalues. It also results in a "bulge" in the upper Hessenberg form of the matrix.

$$H''_{k-1} = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times & \times \\ 0 & + & \times & \times & \times & \times & \times \\ & + & + & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix}$$

Application of Householder reflectors "chase the bulge" down the matrix diagonal.

$$H'''_{k-1} = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times & \times \\ & 0 & \times & \times & \times & \times & \times \\ & 0 & + & \times & \times & \times & \times \\ & & + & + & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix}$$

This process will zero one of the elements on the subdiagonal, which is usually chosen so that the 2×2 matrix in the lower righthand corner is isolated and can be solved analytically. This allows the rest of the matrix to be deflated, reducing the problem. This procedure is iterated until all eigenvalues have been solved.

Results

The Gram-Schmidt orthogonalization kernel performs QR decomposition and has been parallelized successfully. The modified Gram-Schmidt algorithm reduces error and improves convergence as expected. However, certain matrices exist for which the Gram-Schmidt QR method does not converge, such as those which have roots that are complex conjugates of one another.

Discussion

The utilization of the CUDA platform has accelerated the root solving procedure as predicted. Unfortunately, no algorithm yet exists which can universally solve for the eigenvalues of any matrix. However, the physical constraints of the problem may be such that only well-behaved matrices need to be solved. If the Gram-Schmidt QR method proves insufficient, the Francis double-step algorithm allows convergence even for complex conjugate root pairs, so the number of matrices which result in divergent behavior is smaller than with the Gram-Schmidt QR method. The Francis algorithm, along with several tricks to accelerate convergence, is the same one that MATLAB uses to perform the calculations for its `eig()` function. Continuing my work on this project, I will validate the performance of the QR kernel in the complete reconstruction algorithm, refine the parallelization of these algorithms to use GPU resources more efficiently and run more quickly, and parallelize other subroutines to be run on the GPU. I am also investigating the use of wavelets and curvelets in reconstruction of sparse signals.

References

[1] Cao Z, Oh S, Otazo R, et al. Complex Difference Constrained Compressed Sensing Reconstruction for Accelerated PRF Thermometry with Application to MRI-Induced RF Heating. *Magnetic Resonance in Medicine*. 2014.

[2] Farber R. CUDA Application Design and Development. 2011.

[3] Fessler JA. Model-Based Image Reconstruction for MRI. *IEEE Signal Process Mag*. 2010; 27(4):81-89.

[4] Francis JGF. The QR transformation - Parts 1 and 2. *Computer J*. 1961; 4:265-271.

[5] Haack EM. Magnetic Resonance Imaging: Physical Principles and Sequence Design. 1999.

[6] Hansen MS & Sorensen TS. Gadgetron: An Open Source Framework for Medical Image Reconstruction. *Magnetic Resonance in Medicine*. 2013; 69:1768-1776.

[7] Munshi A. OpenCL Programming Guide. 2012.

[8] Parlett BN. The QR algorithm. *Computing Sci. Eng*. 2000; 2:38-42.

[9] Wright KL, Hamilton JI, Griswold MA, et al. Non-Cartesian Parallel Imaging Reconstruction. *Journal of Magnetic Resonance Imaging*. 2014.

A Code

A.1 C++ QR Decomposition Kernel

```
#include <stdio.h>
#include <iostream>
#include <math.h>
#include <cusp/complex.h>
using namespace std;
```

```
typedef cusp::complex<double> cdouble;

__device__
void matrix_print(cdouble *mat, int nDim)
{
   for (int j = 0; j < nDim; ++j)
   {
      for (int i = 0; i < nDim; ++i)
      {
         printf(" %8.3f + %8.3fi", mat[j + i * nDim].real(), mat[j + i * nDim].imag());
      }
      printf("\n");
   }
   printf("\n");
}

__device__
cdouble dotprod(cdouble *vec1, cdouble *vec2, int nDim)
{
   cdouble x = 0;
   cdouble tmp = 0;
   for (int i = 0; i < nDim; ++i)
   {
      tmp.real(vec1[i].real());
      tmp.imag(-vec1[i].imag());
      x += tmp * vec2[i];
   }
   return x;
}

__device__
cdouble* matmult(cdouble *mat1, cdouble *mat2, int nDim)
{
   cdouble *x = new cdouble[nDim * nDim];
   for (int i = 0; i < nDim * nDim; ++i)
      x[i] = 0;
   for (int k = 0; k < nDim; ++k)
      for (int j = 0; j < nDim; ++j)
         for (int i = 0; i < nDim; ++i)
            x[j + k * nDim] += mat1[j + i * nDim] * mat2[i + k * nDim];
   return x;
}

__device__
cdouble l2norm(cdouble *vec, int nDim)
{
   cdouble x = 0;
   for (int i = 0; i < nDim; ++i)
      x += vec[i].real() * vec[i].real() + vec[i].imag() * vec[i].imag();
   x = sqrt(x);
   return x;
}

__device__
void transpose(cdouble *mat, int nDim)
{
   cdouble *x = new cdouble[nDim * nDim];
   for (int i = 0; i < nDim * nDim; ++i)
      x[i] = mat[i];
```

```
   for (int j = 0; j < nDim; ++j)
      for (int i = 0; i < nDim; ++i)
         mat[i + j * nDim] = x[j + i * nDim];
   delete[] x;
}

__device__
void make_comp_mat(cdouble *polynomial, cdouble *companion, int nDim)
{
   for (int i = 0; i < nDim * nDim; ++i)
      companion[i] = 0;
   for (int i = 0; i < nDim; ++i)
      companion[i * nDim] = -polynomial[i + 1] / polynomial[0];
   for (int i = 0; i < nDim - 1; ++i)
      companion[i * nDim + i + 1] = 1;
}

__device__
void select_diag(cdouble *vector, cdouble *matrix, int nDim)
{
   for (int i = 0; i < nDim; ++i)
      vector[i] = matrix[i * nDim + i];
}

__device__
void pixel_mat_select_1d(
   double *a_image_real,
   double *a_image_imag,
   cdouble *a,
   int nDim_matrix,
   int i_image)
{
   int offset_1d = i_image * nDim_matrix;
   for (int i = 0; i < nDim_matrix; ++i)
      a[i] = cdouble(a_image_real[i + offset_1d], a_image_imag[i + offset_1d]);
}

__device__
void pixel_mat_write_1d(
   double *a_image_real,
   double *a_image_imag,
   cdouble *a,
   int nDim_matrix,
   int i_image)
{
   int offset_1d = i_image * nDim_matrix;
   for (int i = 0; i < nDim_matrix; ++i)
   {
      a_image_real[i + offset_1d] = a[i].real();
      a_image_imag[i + offset_1d] = a[i].imag();
   }
}

__device__
void pixel_mat_select_2d(
   cdouble *a_image,
   cdouble *a,
   int nDim_matrix,
   int i_image)
```

```
{
   int offset_2d = i_image * nDim_matrix * nDim_matrix;
   for (int i = 0; i < nDim_matrix * nDim_matrix; ++i)
      a[i] = a_image[i + offset_2d];
}

__device__
void pixel_mat_write_2d(
   cdouble *Q_image,
   cdouble *R_image,
   cdouble *Q,
   cdouble *R,
   int nDim_matrix,
   int i_image)
{
   int offset_2d = i_image * nDim_matrix * nDim_matrix;
   for (int i = 0; i < nDim_matrix * nDim_matrix; ++i)
   {
      Q_image[i + offset_2d] = Q[i];
      R_image[i + offset_2d] = R[i];
   }
}

__device__
void gram_schmidt(cdouble *a, cdouble *Q, cdouble *R, int nDim)
{
   cdouble *u = new cdouble[nDim];
   cdouble *v = new cdouble[nDim];
   cdouble l2 = 0;
   for (int i = 0; i < nDim * nDim; ++i)
      Q[i] = R[i] = 0;
   for (int i = 0; i < nDim; ++i)
      u[i] = v[i] = 0;

   for (int k = 0; k < nDim; ++k)
   {
      for (int i = 0; i < nDim; ++i)
         u[i] = a[i + k * nDim];
      for (int j = k - 1; j >= 0; --j)
         for (int i = 0; i < nDim; ++i)
            u[i] -= R[j + k * nDim] * Q[i + j * nDim];
      l2 = l2norm(u, nDim);
      for (int i = 0; i < nDim; ++i)
         Q[i + k * nDim] = u[i] / l2;
      for (int j = k; j < nDim; ++j)
      {
         for (int i = 0; i < nDim; ++i)
         {
            u[i] = a[i + j * nDim];
            v[i] = Q[i + k * nDim];
         }
         R[k + j * nDim] = dotprod(u, v, nDim);
      }
   }

   delete[] u;
   delete[] v;
}
```

```
__device__
void modified_gram_schmidt(cdouble *a, cdouble *Q, cdouble *R, int nDim)
{
   cdouble *u = new cdouble[nDim];
   cdouble *v = new cdouble[nDim];
   cdouble prj = 0;
   cdouble l2 = 0;
   for (int i = 0; i < nDim * nDim; ++i)
      Q[i] = R[i] = 0;
   for (int i = 0; i < nDim; ++i)
      u[i] = v[i] = 0;

   for (int k = 0; k < nDim; ++k)
   {
      for (int i = 0; i < nDim; ++i)
         u[i] = a[i + k * nDim];
      for (int j = 0; j < k; ++j)
      {
         for (int i = 0; i < nDim; ++i)
            v[i] = Q[i + j * nDim];
         prj = dotprod(v, u, nDim);
         for (int i = 0; i < nDim; ++i)
            u[i] -= prj * v[i];
      }
      l2 = l2norm(u, nDim);
      for (int i = 0; i < nDim; ++i)
         Q[i + k * nDim] = u[i] / l2;
      for (int j = k; j < nDim; ++j)
      {
         for (int i = 0; i < nDim; ++i)
         {
            u[i] = a[i + j * nDim];
            v[i] = Q[i + k * nDim];
         }
         R[k + j * nDim] = dotprod(u, v, nDim);
      }
   }

   delete[] u;
   delete[] v;
}

__device__
void root_find(
   cdouble *polynomial,
   cdouble *root,
   int nDim_in,
   double tolerance,
   int upperbound)
{
   int nDim = nDim_in - 1;
   cdouble *a = new cdouble[nDim * nDim];
   cdouble *Q = new cdouble[nDim * nDim];
   cdouble *R = new cdouble[nDim * nDim];
   int nTol = 0;
   for (int i = 0; i < nDim; ++i)
      root[i] = 0;

   make_comp_mat(polynomial, a, nDim);
```

```
   for (int k = 0; k < upperbound; ++k)
   {
      modified_gram_schmidt(a, Q, R, nDim);
      a = matmult(R, Q, nDim);
      nTol = 0;
      for (int j = 0; j < nDim; ++j)
         for (int i = 0; i < nDim; ++i) {
            if (i > j && sqrt(a[i + j * nDim].real() * a[i + j * nDim].real() +
               a[i + j * nDim].imag() * a[i + j * nDim].imag()) > tolerance)
               ++nTol; }
      if (nTol == 0) break;
   }

   select_diag(root, a, nDim);

   delete[] a;
   delete[] Q;
   delete[] R;
}

__global__
void QRDRoot(
   double *polynomial_image_real,
   double *polynomial_image_imag,
   double *root_image_real,
   double *root_image_imag,
   double const tolerance,
   int const upperbound,
   int const nDim_image,
   int const nDim_matrix)
{
   int i_image = blockDim.x * blockIdx.x + threadIdx.x;
   if (i_image > nDim_image * nDim_image) return;

   cdouble *polynomial = new cdouble[(nDim_matrix) * (nDim_matrix)];
   cdouble *root = new cdouble [(nDim_matrix - 1) * (nDim_matrix - 1)];

   pixel_mat_select_1d(polynomial_image_real, polynomial_image_imag, polynomial,
      nDim_matrix, i_image);
   root_find(polynomial, root, nDim_matrix, tolerance, upperbound);
   pixel_mat_write_1d(root_image_real, root_image_imag, root, nDim_matrix - 1, i_image);

   delete[] polynomial;
   delete[] root;
}
```

A.2 MATLAB Driver

```
%function driver

clear all
close all
format shortg

tol = 0.0001;
upbound = 1000;
```

```
nDim_image = 2;
nDim_matrix = 4;

h_root = complex(zeros(nDim_matrix-1,nDim_image,nDim_image),zeros(nDim_matrix-1,nDim_image,
    nDim_image));
h_poly = complex(randn(nDim_matrix,nDim_image,nDim_image),randn(nDim_matrix,nDim_image,nDim_image))
    ;
%h_a = randn(nDim_matrix,nDim_matrix,nDim_image,nDim_image);
%h_Q = zeros(nDim_matrix,nDim_matrix,nDim_image,nDim_image);
%h_R = zeros(nDim_matrix,nDim_matrix,nDim_image,nDim_image);

%for i = 1:nDim_image
%    for j = 1:nDim_image
%        h_a(:,:,i,j) = [1,1,0;1,0,1;0,1,1];
%    end
%end

%for i = 1:nDim_image
%    for j = 1:nDim_image
%        h_poly(:,i,j) = [1,-6,-72,-27];
%    end
%end

% transfer data to device
d_poly = gpuArray( h_poly );
d_root = gpuArray( h_root );
%d_a  = gpuArray( h_a );
%d_Q  = gpuArray( h_Q );
%d_R  = gpuArray( h_R );

qrdptx = parallel.gpu.CUDAKernel('qrd.ptx', 'qrd.cu');
threadsPerBlock = 256;
npixel = nDim_image*nDim_image;
qrdptx.ThreadBlockSize=[threadsPerBlock, 1, 1];
%blocksPerGrid = (npixel + threadsPerBlock -1) / threadsPerBlock;
%blocksPerGrid = (npixel * threadsPerBlock - 1) / threadsPerBlock;
blocksPerGrid = ceil(npixel/threadsPerBlock);
qrdptx.GridSize=[blocksPerGrid, 1, 1];
%qrdptx.GridSize=[ceil(blocksPerGrid), 1, 1];
%qrdptx.GridSize=[256, 1, 1];

[dpolyrealout,dpolyimagout,drootrealout,drootimagout] = feval(qrdptx,real(d_poly),imag(d_poly),real
    (d_root),imag(d_root),tol,upbound,nDim_image,nDim_matrix);
%[daout,dQout,dRout] = feval(qrdptx,d_poly,d_Q,d_R,nDim_image,nDim_matrix);

hf_poly = gather(complex(dpolyrealout,dpolyimagout));
hf_root = gather(complex(drootrealout,drootimagout));

for i=1:nDim_image
   for j=1:nDim_image
      matsol(:,i,j)=roots(hf_poly(:,i,j));
      xcheck=norm(hf_root(:,i,j)-matsol(:,i,j))
   end
end

%for i=1:nDim_image
%    for j=1:nDim_image
%        xtest(:,i,j)=h_A(:,:,i,j)\h_b(:,i,j);
%        xcheck = norm(xtest(:,i,j)-dxout(:,i,j));
```

```
%        if xcheck >= 1.0e-8
%            xcheck
%        end
%    end
%end

%exit
```