Versatile higher-order reconstruction accelerated by a graphics processing unit (GPU)

M. A. Bieri[1], C. Barmet[1], B. J. Wilm[1], and K. P. Pruessmann[1]
[1]Institute for Biomedical Engineering, University and ETH Zurich, Zurich, Zurich, Switzerland

Introduction: The vast majority or MR images is currently sampled on a Cartesian k-space grid and can therefore be efficiently reconstructed by Fast Fourier Transforms (FFT). In practice however, Fourier encoding is often complemented by other encoding mechanisms, such as static B_0 off-resonance fields, motion, concomitant fields, Eddy currents of several spatial orders. Furthermore, B1-sensitivity encoding and non-Cartesian k-space sampling are commonly combined encoding strategies. For many of these reconstruction problems specific algorithms have been developed. When combining several of these methods, the resulting algorithms become increasingly complex or inefficient. Alternatively an algebraic reconstruction can be used, that allows the incorporation of various encoding mechanisms with only minor increase in complexity as compared to its 'basic' Fourier implementation. However, even for small images of about 100x100 pixels, reconstruction times range on the order of minutes on current high end CPUs and scale with the fourth power of the image width for quadratic images. Modern GPU architectures consist of several hundred programmable floating point processors, lending themselves for highly parallel problems such as algebraic MR image reconstruction. Gain in computational speed compared to a CPU is evaluated for the intricate example of parallel imaging combined with the correction for dynamic higher-order field effects [1] and static B_0, demonstrating the versatility of the proposed reconstruction algorithm.

Methods: *Algorithm:* The image reconstruction is implemented by solving the linear system given by Eq.1 for the unknown image ρ, given the signal σ and using a conjugate gradient (CG) algorithm. $\boldsymbol{r}$ is the coordinate vector of image points, $\boldsymbol{t}$ is the time vector of the signal samples. λ, κ and ν are spatial, time and coil indices, respectively. The encoding matrix $\mathbf{E}$ incorporates B_1-sensitivity encoding $\boldsymbol{s}$, static B_0 encoding (ΔB_0) and dynamic higher order fields (Eq.2). The dynamic higher order terms are given as an expansion into spherical harmonics (including k_x, k_y and k_z as well as higher-order terms [1]) of the magnetic field (Eq.3). A set of $m = 16$ real-valued spherical harmonics $h_m(r)$ were used. Coefficients $k_m(t)$ were measured by a concurrent magnetic field monitoring setup [2]. Typically the number of image pixels is similar to the number of samples acquired in time (x number of coils). $\mathbf{E}$ therefore scales with $O(n^2)$ in image pixels or $O(n^4)$ in image width for a quadratic image.

$$\mathbf{E}^H\mathbf{E}\,\rho(r_\lambda) = \mathbf{E}^H\sigma(t_\kappa) \quad (1)$$

$$\mathbf{E}_{(\nu,\kappa),\lambda} = \mathbf{s}_\nu(\mathbf{r}_\lambda)\exp(i\cdot\phi(\mathbf{t}_\kappa,\mathbf{r}_\lambda)) \quad (2)$$

$$\phi(t,\mathbf{r}) = \Delta B_0(r)\cdot t + \sum_m k_m(t)h_m(\mathbf{r}) \quad (3)$$

Fig. 1: Equations defining the reconstruction

Hardware: Computing was performed on an Nvidia GeForce GTX 480. The CUDA [3] environment is used as a programming interface. A CUDA program is defined as a so-called *kernel*. This program is executed parallel by a large number of identical *threads,* grouped in *thread blocks*. Each thread block runs on one *streaming multiprocessor* (SM). A SM contains 32 floating point and integer processors called *CUDA cores* and a (fast) *shared memory* which is common to all threads in a block. The graphics processor consists of 15 SM and is attached to a (relatively slow) *global memory* of 1.5 GB. Communication with the *host system* (and therefore any I/O) is performed via a PCI-E bus (Fig.1).

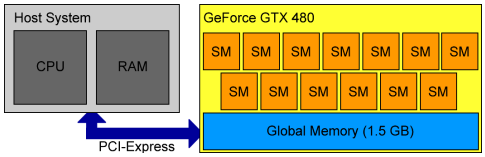

Fig. 2: GeForce GTX 480 and host system.

Implementation: The algorithm was implemented in C for CUDA and provides a Matlab MEX interface. The input data is transferred from the host computer to the graphics card global memory at the beginning and output is read back at the end of the reconstruction. The entire algorithm runs on the GPU to reduce the transfer via the (slow) PCI-E bus to a minimum.

The computationally most complex part of the CG algorithm are the two matrix-vector computations $\mathbf{v} = \mathbf{E}^H\mathbf{u}$ and $\mathbf{u} = \mathbf{E}\mathbf{v}$. Since $\mathbf{E}$ is too big to be stored in the global memory or even in the small on-chip shared memory, the elements of $\mathbf{E}$ have to be computed each time they are needed. 16 GB of memory are required by the encoding matrix of a 256x256 image, 1 coil (each element consists of two 32 bit floating point numbers, real and imaginary part). Another reason not to store $\mathbf{E}$ globally is the relatively slow transfer speed of global memory. Roughly estimated, one memory transfer of a floating point number costs as much as 30 floating point operations on the GPU (1.3 TFLOPS vs. 177 GB/s = $45\cdot10^9$ floating point numbers/s memory transfer).

To preserve computing time and memory bandwidth, each thread in the kernels for $\mathbf{E}^H\mathbf{u}$ and $\mathbf{E}\mathbf{v}$ first load and compute some parts of the input and $\mathbf{E}$ and then shares with all threads of the same block via the fast shared memory.

Test: Two different data sets are used for testing. To evaluate computation time and accuracy, MR data of a Shepp-Logan phantom was simulated assuming Cartesian k-space sampling including typical (measured) higher-order phase coefficients, sensitivity maps and a simulated ΔB_0 map. Simulated data is needed for accuracy testing and freely scalable input size (for speed testing). The benefit of the higher order method is demonstrated for a diffusion-weighted single shot EPI dataset with 3-fold undersampling. 50 iterations were performed for each reconstruction to ensure convergence.

Images were reconstructed comparing different hardware platforms: First, a MATLAB implementation of the algorithm running on an Intel Core 2 Duo 3.0 GHz. This implementation spends about 90% of the computation time in MATLAB built-in matrix operations and is therefore expected to be comparable to a native CPU implementation. Second, the presented CUDA implementation using a GTX 480. Third, the CUDA implementation is tested on a cheap (80US$, fall 2010) Nvidia GeForce 9600 GT.

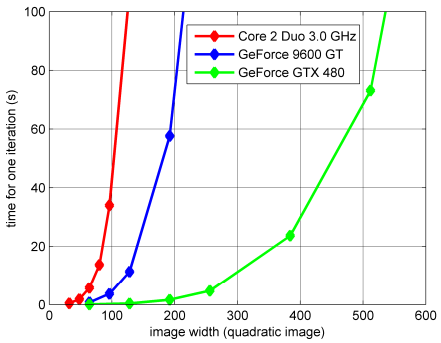

Fig. 3: Computing time vs. image width for quadratic image reconstruction with 16 higher orders, 8 coils, static B0 correction and quadratic encoding matrix.

Results and Discussion: Computation times are given in Fig.3. For a 512x512 image, the Core 2 Duo takes 12 hours 51 minutes while the GTX 480 needs 73 seconds. This implies a ***speed-up of 633x***. Speed-up decreases with problem size, for 256x256 and 64x64 they are 581x and 216x, respectively. The cheap 9600 GT needs 182 seconds for 256x256 images, representing a speedup of 15x against the CPU.

Average and maximum errors of the CUDA implementation are on the same order of magnitude as in the MATLAB implementation. The relative maximum error is about 10^{-3}, the average error is about 10^{-6}.

The absolute error against a conventional reconstruction with the linear model is shown in Fig. 4b. As a benefit of incorporating the more complex encoding model, errors associated from higher-order fields (Fig. 4b) can be straightforwardly corrected (Fig. 4a).

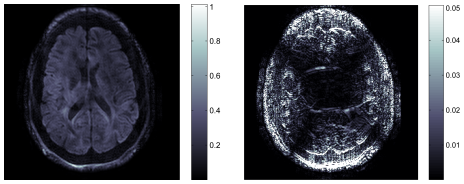

Fig.4: In-vivo diffusion-weighted image reconstructed using (a) the higher-order field model and the difference image (b) when neglecting higher-order fields.

Conclusion: Using the presented implementation and GPU hardware, higher-order image reconstruction was sped-up by orders of magnitude as compared to a comparable reconstruction on a CPU. Further acceleration can be expected in the future with rapidly increasing graphics card performance.

References: [1] Wilm et al., MRM (in press) [2]: Barmet et al. ISMRM2010, 216. [3] Nvidia Corp. 2010, http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf