BLOCH SOLVER SIMULATION REALIZATION ON A GRAPHICS-PROCESSING UNIT (GPU)

S. M. Lechner[1,2], D. Butnaru[2], H-J. Bungartz[2], D. Chen[1,3], and M. W. Vogel[1]

[1]Advanced Medical Applications Laboratory, GE Global Research, Munich, Bavaria, Germany, [2]Department of Scientific Computing in Computer Science, Technical University Munich, Munich, Bavaria, Germany, [3]Department of Scientific Computations, Technical University Munich, Munich, Bavaria, Germany

Introduction:

Simulating the spin behavior is key to different research areas within the field of magnetic resonance imaging (MRI). Solving the Bloch equations of large volumes of interest, simulations are limited because of the high number of evaluations in relation to achievable computational speed and available memory. In the past, several approaches have been reported where the Bloch equations are solved in parallel on computer clusters realizing algorithm implementations based on the message-passing interface (MPI) [1,2]. However, computer clusters are highly expensive and often built of a variety of different hardware components. Expertise and expert programming skills are needed to transfer the software into a parallel environment that is capable of fully utilize the given hardware performance. Nowadays, modern GPUs offer affordable high computational power. Due to the existent user-friendly programming interface, concrete knowledge about computer graphics is not necessary anymore in order to obtain improved calculation speed. For this reason, a fast, robust, flexible and parallel realization of the Bloch solver has been transferred on a GPU [3]. The discrete time solutions of the Bloch equations are calculated for each spin in parallel resulting in speed up times that are 6-12 times faster than our fast single processing implementation.

Materials and Methods:

The GPU is considered as computational device and is especially designed for computational intensive and highly parallel computations. The computer unified device architecture (CUDA [4]) provides a high-level C-like programming interface that allows a kernel functionality to be massively parallelized. Optionally, the processes interact through particular communication routines. Running a CUDA application, the data gets converted to single precision. The data is then copied to the GPU memory and the compiled and offloaded kernel function is executed, where the results are delivered back from GPU memory into local memory. A similar approach is shown in Ref. [5], where GPUs are utilized to speed up image reconstruction algorithms. Within MRI simulations, the spins are independent from each other. Hence, the volume of interest is split into several blocks of equal size. Each block is assigned to a processor, where a number of threads process each spin of the block while copying and executing the kernel function. Each processor has its own global and shared memory. Fig.1 summarizes the CUDA solver execution procedure including pre- and post processing steps and the time loop iterations that need to be executed by each thread. Alternatively, the solver module is exchangeable, for example the performance of the CUDA solver is compared to two other realizations, the Looping Solver (LS) or Vectorization Solver (VS). LS is a non-parallelized version of the CUDA realization including a (3D) looping structure over each spin within the volume of interest (x, y and z respectively). The Bloch equations are solved for each time step individually. VS is based on existent Matlab vectorization techniques and functions [6], e.g. logical or vector based matrix indexing.

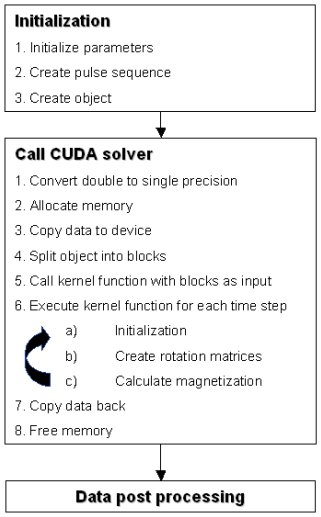

Fig1: Schematic overview of CUDA solver execution process.

Results:

The CUDA Bloch solver was implemented on a GeForce 8800 GT (Nvidia, Santa Clara, CA) graphics card with 512MB of memory on a PC with 8 GB of memory and a 3.20 GHz processor running Windows XP x64 edition. The GPU code was written in C and compiled using CUDA together with Visual C++ 2005 (Microsoft Corporation, 2006). The CPU reference algorithms are both implemented and executed using Matlab (The Mathworks Inc., Cambridge, MA, Version R2007b). Table 1 summarizes the solver times in seconds for the three different solver implementations. For all experiments, a gradient echo (GE) sequence was used and run on four different volume resolutions with equal time stepping. LS could be executed in acceptable computational times till matrix size 32x32. The LS algorithm has a complexity of *resolution² x time*. Doubling or quadrupling the object resolution results in a scaling factor of 4 or 16 respectively. These scaling factors were used to estimate solving times for object sizes of 64x64 and 128x128 in case of LS. For GE imaging, we achieved speed up factors of 12.2 and 870.7 for LS/CUDA and VS/CUDA respectively. Increasing the object size resulted in constant factors for LS/CUDA, but halved the factors for VS/CUDA probably due to too small compared to too large vector sizes. Comparing the results from [2], the simulation process was speed up by factors 2 and 6 in cases 64x64 and 128x128. Fig.2a shows the received free induction decay (FID) of the GE experiment for VS (blue), LS (green stars) and CUDA (red dashed line). VS and LS fully overlap with an accuracy of 10^{-10}, whereas the CUDA solver slightly differs in accuracy to a magnitude of 10^{-5} in both cases, which can be explained by the rounding error accumulation for double and single calculations at each time step.

Matrix size	LS [s]	VS [s]	CUDA Solver [s]	LS vs. CUDA	VS vs. CUDA
16^2x3408	85.68	1.20	0.10	870.70	12.20
32^2x3472	348.92	2.36	0.41	851.02	5.76
64^2x3600	1395.68	9.89	1.63	856.25	6.07
128^2x3856	5582.72	45.66	7.11	785.19	6.42

Table1: Summary of the data evaluation based on three different Bloch solver implementations: LS based on looping structure, VS which takes advantage of Matlab vectorization tools and CUDA, the GPU realization. CUDA is compared with LS and VS respectively. The speed-up factors are listed in the last two columns.

Discussion and Conclusion:

Using the GPU-accelerated implementation of the Bloch solver significantly speeds up the simulation process compared to our fast CPU reference implementation. The GPU realization is a reasonably prized and compact alternative to computer clusters. Parallel programs can be readily embedded within the predefined CUDA environment and deliver high-resolution images at reasonable times. Profiling the implemented algorithms showed that most time is spent in reading/copying data from/to memory, memory allocation or additional conversions (67%), whereas 33% of the solver time is used for the real calculations. In our setup, each processing unit has 256 threads available to process the kernel function. Creating blocks of the size 256 and having 32 registers and 16 kilobytes of shared memory available for each thread gained best occupancy (33%) for each processor. This shows that the major performance penalty is due to accessing memory. Each thread is individually copying all the needed data into memory although some data is reusable for other spins. In addition, results are written back into memory within the kernel function also resulting in slower execution times. Organizing memory and synchronizing the running threads decreases memory accesses and might speed up the solver. The presented solver is embedded in custom Matlab pre- and post- processing routines adding additional computational time. Transferring the full simulation environment into the parallelized GPU approach and extending the used hardware to a set of different graphics cards, e.g. a Tesla Box, with more memory and processors may help to further improve the speed advantage of the GPU implementation.

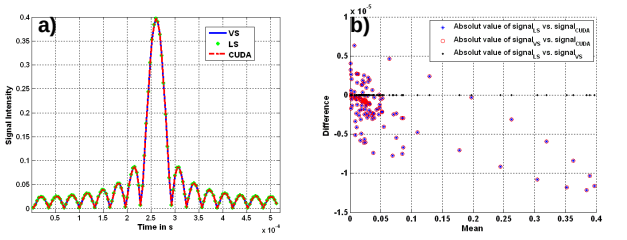

Fig2a-c: Received FID for LS (green), VS (blue) and CUDA (red) and a comparison of them showing the Bland-Altmann plot of the absolute value of the signal in a).

References:

[1] Brenner et. al., ISMRM 1997, #2052, [2] Benoit-Cattin et. al., JMR 173, p. 97, 2005,
[3] http://www.nvidia.com/page/geforce_8800.html, Oct. 08, [4] http://www.nvidia.com/object/cuda_home.html#, Oct. 08,
[5] Sørensen et. al., IEEE Tran. Med. Im. 27, No.4, p. 538, 2008, [6] http://www.mathworks.com/support/tech-notes/1100/1109.html, Oct. 08