Real-time Non-rigid Registration of Medical Images on a Cooperative Parallel Architecture

Yixun Liu[1], Andriy Fedorov[2], Ron Kikinis[2], Nikos Chrisochoides[1]
[1]*Department of Computer Science, College of William and Mary, Williamsburg, USA*
[2]*Surgical Planning Laboratory, Brigham and Women's Hospital, Boston, USA*
enjoywm@cs.wm.edu, {fedorov,kikinis}@bwh.harvard.edu, nikos@cs.wm.edu

Abstract—**Unacceptable execution time of Non-rigid registration (NRR) often presents a major obstacle to its routine clinical use. Parallel computing is an effective way to accelerate NRR. However, development of efficient parallel NRR codes is a very challenging task. One desirable approach is to map the existing sequential algorithm to the parallel architecture to gain speedup instead of designing a new parallel algorithm. Multicores and GPU provide us a cooperative architecture, in which both Single Instruction Multiple Data (SIMD) and Single Program Multiple Data (SPMD) programming models can co-exist and complement each other. We present a method to parallelize a NRR on this cooperative architecture. Our approach is first to separate the sequential algorithm into regular and irregular parts. We then map the regular part on GPU following SIMD paradigm and irregular part on multicores in a SPMD fashion. Unlike the approaches that use multicores or GPU alone, our approach leads to desirable speedup for the whole application by taking advantage of all components of the cooperative parallel architecture, for all individual parts of the application. This helps us to get closer to our goal: cheaper and faster NRR that leads to its more widespread use. The results on clinical brain MRI data show that the GPU-based Block Matching (regular part) can run at least 1.9 times faster than on a typical cluster of workstations with eight high-performance nodes. The multicores-based implementation of the incremental finite element solver (irregular part) achieves speedup of up to 7 times compared to its sequential version. As a result, the total run time of the NRR code can be reduced to less than 1 minute therefore satisfying the real time requirement for its clinical application.**

Keywords-**Non-rigid registration, Block matching, Incremental finite element solver, GPU, Multicores**

I. Introduction

Image registration is the process of aligning images so that the corresponding features can be easily related. Image registration falls into two categories: Rigid registration and Non-rigid registration (NRR). The long CPU runtime of the existing NRR techniques is a major barrier to their routine clinical use in all time-critical intra-operative applications.

A host of studies employ parallel computing to accelerate NRR on multicores or clusters [1], [2], [3], [4]. Recently, some groups implemented it on Graphics Processing Units (GPUs) [5], [6], [7], [8]. However, up to now there were no attempts to accelerate NRR using the cooperative architecture: multicores and GPU, which is widely available in commodity PCs. This cooperative architecture can be easily deployed in the Operating Room (OR) without hindering routine surgery procedures. In addition, GPU's SIMD programming model and multicores' SPMD programming model complement each other and provide a powerful and flexible hybrid programming environment. GPU has massive number of cores, which can effectively deal with regular computations, but provides limited support for communication and synchronization [9]. Multicores architectures have limited cores, but are more universal allowing to implement irregular algorithms, and providing flexible support for communication and synchronization [10]. An interesting study about strong and weak sides of multicores vs. GPU have been recently presented by Intel research [10]. Algorithms with following characteristics are defined as irregular algorithms.

- Require dynamic data types, such as sparse matrices, linked lists, or trees;
- Have high likelihood of contended synchronization;
- Have moderate control flow, such as well-structured conditional nests, nested loops, and recursive functions.

Intel multicores architecture (including tera-scale architecture) addresses these requirements efficiently, whereas GPU generally does not.

If we separate NRR into regular part and irregular part and implement them on GPU and multicores respectively we can gain desirable speedup for each part. Moreover, with little modification to the original algorithm, each part can be easily mapped to its corresponding programming model, as we show in this paper. Minimizing changes to the original algorithm is very important. Usually, before we resort to parallel computing to accelerate our application, we have a well-designed sequential code, with established accuracy, robustness and performance. As we parallelize it, we are trying to map this sequential code to a parallel architecture to gain speedup instead of developing the parallel algorithm from the scratch. The cooperative architecture makes this

This material is based upon work supported by the National Science Foundation under Grants No. CCF-0916526, CCF-0833081, CSR-0719929, and CCS-0750901, NIH under Grants No. P41 RR13218, U54 EB005149, U41 RR019703, and John Simon Guggenheim Foundation.

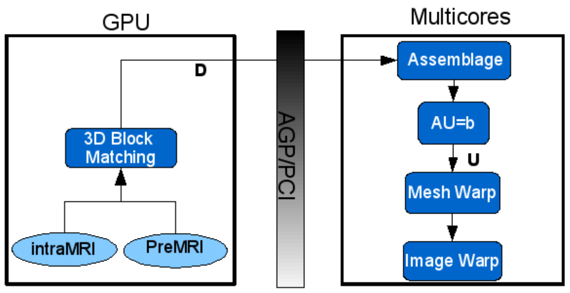


Figure 1. Parallel NRR framework. Left: GPU implementation of Block Matching. Right: multicores implementation of incremental FE solver

possible. To the best of our knowledge, the only study to utilize this cooperative architecture was presented by Hartley et al. [11] in the context of medical image analysis. However, the authors did not address NRR in their work.

The contributions of this paper are twofold: (1) A GPU based 3D Block Matching algorithm (regular part); and (2) A multicores based incremental finite element solver (irregular part). Based on these two parts, a parallel implementation of NRR on the cooperative architecture is presented, which is characterized by (1) Desirable speedup and (2) Minimal changes to the existing sequential algorithm.

II. Non-Rigid Registration Approach

The NRR method we are targeting is based on the concept of energy minimization [12]. A sparse set of registration points within the preoperative brain MRI are identified. The displacement between the pre- and intraoperative images is estimated using Block Matching [13] at each registration point. Based on these displacements, the deformation field defined at mesh nodes is estimated under the constraint of a biomechanical model.

Registration is formulated as an energy minimization problem:

$$\mathbf{U} = \underset{\mathbf{U}}{\operatorname{argmin}}\{(\mathbf{HU}-\mathbf{D})^T\mathbf{S}(\mathbf{HU}-\mathbf{D}) + \mathbf{U}^T\mathbf{KU}\}, \quad (1)$$

where $\mathbf{K}$ is the stiffness matrix, $\mathbf{H}$ is the linear interpolation matrix from the displacements recovered by Block Matching (BM) and those at the mesh vertices, $\mathbf{S}$ is the BM weight matrix, $\mathbf{D}$ contains BM displacements, and $\mathbf{U}$ is the unknown displacement vector at the mesh vertices.

Regularization of the solution using the mechanical energy is susceptible to outliers and unavoidably contains approximation error [12]. We address this problem by iterative estimation of the displacement:

$$\mathbf{F}_0 = 0, \ \mathbf{F}_{i-1} = \mathbf{KU}_{i-1},$$
$$[\mathbf{K} + \mathbf{H}^T\mathbf{SH}]\mathbf{U}_i = \mathbf{H}^T\mathbf{SD} + \mathbf{F}_{i-1} \quad (2)$$

This iterative method reduces the approximation error at each iteration, while rejecting outliers. In the reminder of the paper we call equation 2 *incremental Finite Element (FE) Solver* due to its incremental improvement of the accuracy. Such formulation is robust against outliers and minimizes the approximation error at the expense of longer execution time.

The NRR method contains two computationally intensive components: Block Matching and the incremental Finite Element Solver. Our parallel NRR framework is shown in Fig. 1. Block Matching is characterized by regular data and regular operations, therefore we use GPU designed to perform bulk computations of a kernel code on different input data. The incremental Finite Element Solver operates on irregular data structures requiring synchronization and communication. Such computations cannot fully benefit from GPU architecture [10], therefore we develop multicore implementation of the solver. A nonlinear FE solver has been implemented by [14] on GPU. Compared to the GPU implementation, multicore SPMD model allows higher level of parallelism. We can take full advantage of the existing sequential code to parallelize it *horizontally* instead of *vertically* by some preprocessing: data partitioning and renumbering, which will be discussed in the following section.

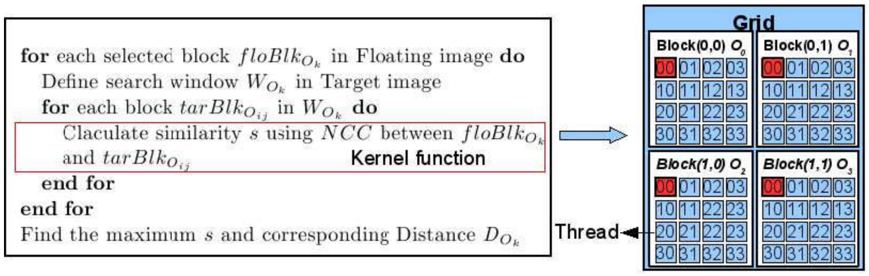
III. Parallel Implementation

In this section we present the GPU implementation of 3D Block Matching. Data partitioning will be used to parallelize incremental FE Solver under the constraint of minimizing communication between processors. Global and local renumbering of vertices are employed to improve performance and facilitate mapping from sequential code to parallel architecture respectively.

A. GPU Implementation of 3D Block Matching

Block Matching is a well known technique used originally for recovering motion from images [13]. BM is based on the assumption that a complex non-rigid transformation can be approximated by point-wise translations of small image regions. Such a translation can be recovered at a point of the floating image by selecting a block of voxels $B(O_k)$ centered around a point O_k, and searching for such a displacement vector that maximizes some similarity metric $M(B(O_a), B(O_b))$ with respect to the corresponding part of the target image window W_k. The similarity metric depends on the application, with Normalized Cross Correlation (NCC) suitable for registering mono-modal data.

The key to efficient GPU implementation is in the mapping of a sequential program to CUDA programming model, which is demonstrated in Fig. 2. Due to the regularity of BM, the mapping is straightforward: the outer loop is mapped to GPU Grid; the inner loop is mapped to GPU thread block and NCC computation is mapped to the GPU kennel function. More specifically, for each registration point and the corresponding block in the floating image, we assign a separate CUDA thread to calculate its similarity with a different fraction of the search window. The size of the fraction depends on the size of the thread block



```
for each selected block $floBlk_{O_k}$ in Floating image do
    Define search window $W_{O_k}$ in Target image
    for each block $tarBlk_{O_{ij}}$ in $W_{O_k}$ do
        Claculate similarity $s$ using $NCC$ between $floBlk_{O_k}$
        and $tarBlk_{O_{ij}}$                Kernel function
    end for
end for
Find the maximum $s$ and corresponding Distance $D_{O_k}$
```

Figure 2. Mapping from sequential BM to GPU programming model. Left: Sequential BM. Right: GPU BM.

and the search window. For instance, if the thread block is $4 \times 4 \times 4$ and search window is $8 \times 8 \times 8$, each thread will be responsible for the calculation of the similarity within a $2 \times 2 \times 2$ window. The maximum similarity can be evaluated by parallel reduction of the computed similarity values. As we map NCC calculation to kernel function, the only change we make is to access data from GPU texture memory instead of CPU memory. For instance, $CPUArray[i][j][k] \Rightarrow tex3D(GPUTexture, k, j, i)$.

B. Multicores Implementation of Incremental Solver

The matrices used in equations 2 are derived from finite element mesh and registration points. The objective of the parallelization is to distribute the mesh and registration points among the processors, and derive the solution to the linear system of equations in parallel.

We employ the ParMETIS library [15] to implement balanced parallel partitioning of the tetrahedral mesh among the processing cores. ParMETIS starts with an initial partitioning of the input mesh. Based on the initial partitioning, ParMETIS generates a mapping between the elements and the processors to minimize the number of the interface elements.

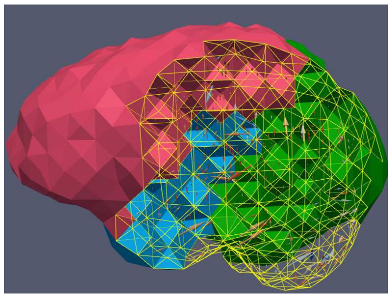


Figure 3. 4-way partitioned mesh and registration points.

Following the partitioning, vertices, i.e. unknowns, in each of the sub-meshes must be renumbered contiguously. We use local numbering strategy and let each processor keep a mapping table to relate the local numbering with the global numbering. The advantage of this approach is that the sub-mesh can be considered as a separate mesh on each of the processors, while the communication with the non-local sub-meshes is facilitated by the mapping table. With this mapping table, replacing the access to $Node[i]$ with $Node[MappingTable[i]]$ the original sequential code is easily to be parallelized on multicores. After renumbering we can assemble the linear system of equations and solve it using PETSc [16].

IV. Performance Results

A. Block Matching Results

We compared the performance of the Block Matching on a typical modern workstation equipped with NVIDIA GeForce 8800 GT GPU with its MPI implementation running on a 8-node cluster (each node is Dell PowerEdge SC1435, 2 x dual-core Opteron 2218, 2.6 GHz CPU). The results were collected for computations on 6 retrospective brain tumor resection cases, with the imaging parameters similar to the ones used for acquisition of brain imaging in SPL Brain Tumor Resection dataset[1].

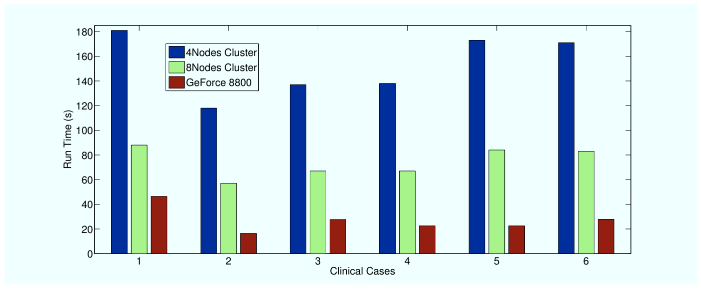


Figure 4. Performance evaluation using six existing retrospective data from BWH. Image block: $9 \times 9 \times 9$, Search window: $11 \times 11 \times 19$, Thread block: $4 \times 4 \times 4$.

Fig. 4 shows the comparison of performance for the considered implementations. Compared to the 4-node cluster (16 CPUs), the minimum speedup is 3.9 (case 1) and the maximum speedup is 7.7 (case 5). Compared to the 8-node cluster (32 CPUs), the minimum speedup is 1.9 (case 1) and the maximum speedup is 3.8 (case 5).

B. Incremental Solver Results

The solver runtime depends on the size of the mesh. Three meshes of increasing sizes were generated. Biconjugate gradient solver [17] is used for comparison with our parallel incremental solver. The execution times for assembling and iterative solution of the linear system are listed in Table I.

[1]http://www.spl.harvard.edu/publications/item/view/541

Table I
PERFORMANCE COMPARISON BETWEEN SEQUENTIAL AND PARALLEL SOLVER

Mesh		Sequential(time:second)		Parallel(time:second)			Speedup
Vertices	Tets	Assemblage	Solver	Partition	Assemblage	Solver	
1607	7272	6.780	23.560	0.820	0.760	1.900	8.718
3526	17137	7.500	31.280	1.880	0.79	2.37	7.694
6737	33931	8.040	43.520	3.81	0.78	3.32	6.518

Compared with sequential solver, our parallel solver needs additional partitioning time, but the overall gain in performance is significant. We evaluate our parallel implementation on tumor resection cases, the registration results are shown in Fig. 5.

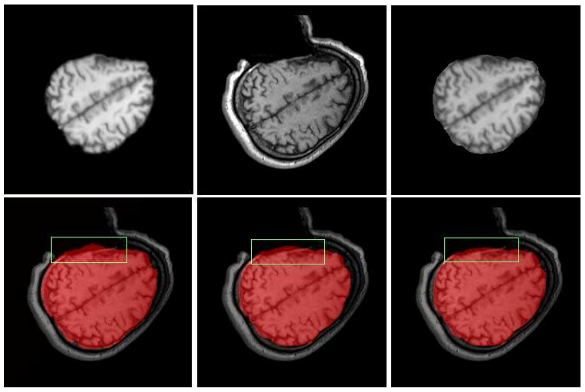


Figure 5. Registration results. Top left: preoperative MRI. Top middle: intraoperative MRI. Top right: deformed preoperative MRI. Bottom left: deformed preoperative MRI (red) superimposed on intraoperative MRI. Bottom middle: deformed preoperative MRI(1st iteration) superimposed on intraoperative MRI. Bottom right: deformed preoperative MRI (10th iteration) superimposed on intraoperative MRI.

Each part gains desirable speedup reducing the total run time to less than 1 minute (calculated using large mesh in TableI for case 1 in Fig.4: $45 + 3.81 + 0.78 + 3.32 = 52.91s$). The algorithms in both regular and irregular parts are unchanged, maintaining the accuracy of the sequential code. Our experiments (see Fig. 6) show that the difference of the accuracy between the parallel and the sequential implementations is below 0.006mm (large mesh), which is normal for parallel implementation due to concurrency.

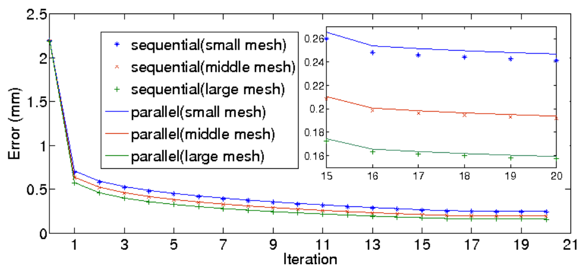


Figure 6. Precision comparison between sequential and parallel NRR on different size of meshes.

V. CONCLUSIONS

As we parallelize the existing sequential algorithm, we expect to accelerate it using available parallel architecture instead of redesigning it from the scratch in order to retain the accuracy and robustness of the sequential code evaluated in clinic. We presented a parallel implementation based on the cooperative architecture and show how to best utilize both GPU and multicores for the parallelization of an existing sequential FEM based NRR algorithm. Our approach separates the sequential NRR into regular part and irregular part and implements them on GPU and multicores respectively. The mapping of the regular part to GPU programming model is straightforward. By means of data partitioning and renumbering strategy the irregular part of the sequential code can be easily mapped to multicores. Compared to 8-node cluster our parallel NRR can reduce the execution time to less than 1 minute on a 11 times cheaper workstation.

REFERENCES

[1] F. Ino, K. Ooyama, and K. Hagihara, “A data distributed parallel algorithm for nonrigid image registration,” *Parallel Computing*, vol. 31, no. 1, pp. 19 – 43, 2005.
[2] T. Rohlfing and C. Maurer, “Nonrigid image registration in shared-memory multiprocessor environments with application to brains, breasts, and bees,” *IEEE Transactions on Information Technology in Biomedicine*, vol. 7, no. 1, pp. 16 – 25, 2003.
[3] M. Sermesant, O. Clatz, Z. Li, S. Lantri, H. Delingette, and N. Ayache, “A parallel implementation of non-rigid registration using a volumetric biomechanical model,” in *WBIR*, 2003, pp. 398–407.
[4] N. Chrisochoides, A. Fedorov, A. Kot, N. Archip, P. Black, O. Clatz, A. Golby, R. Kikinis, and S. Warfield, “Toward real-time image guided neurosurgery using distributed and Grid computing,” in *Proc. of IEEE/ACM SC06*, 2006.
[5] A. Ruiz, M. Ujaldon, L. Cooper, and K. Huang, “Non-rigid registration for large sets of microscopic images on graphics processors,” *J Sign Process Syst*, vol. 55, no. 1-3, pp. 229–250, April 2008.
[6] C. Vetter, C. Guetter, C. Xu, and R. Westermann, “Non-rigid multi-modal registration on the GPU,” in *Medical Imaging 2007: Image Processing*, vol. 6512, 2007, p. 651228.
[7] P. Muyan-Ozcelik, J. Owens, X. Junyi, and S. Samant, “Fast deformable registration on the GPU: A CUDA implementation of Demons,” 2008, pp. 223–233.
[8] R. Yousfi, G. Bousquet, and C. Chefd’hotel, “New GPU optimizations for intensity-based registration,” in *Proceedings of the SPIE - The International Society for Optical Engineering*, vol. 7259, 2009.
[9] NVIDIA, “Cuda programming guide2.0,” 2008.
[10] Intel Research, “Ct: A flexible parallel programming model for tera-scale architectures,” http://download.intel.com/pressroom/kits/research/Flexible_Parallel_Programming_Ct.pdf.
[11] T. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, “Biomedical image analysis on a cooperative cluster of GPUs and multicores,” in *ICS ’08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 15–25.
[12] O. Clatz, H. Delingette, I.-F. Talos, A. Golby, R. Kikinis, F. Jolesz, N. Ayache, and S. Warfield, “Robust non-rigid registration to capture brain shift from intra-operative MRI,” *IEEE Trans. Med. Imag.*, vol. 24, no. 11, pp. 1417–1427, 2005.
[13] M. Bierling, “Displacement estimation by hierarchical block matching,” *Proc. SPIE Vis. Comm. and Image Proc.*, vol. 1001, p. 942951, 1988.
[14] Z. Taylor, M. Cheng, and S. Ourselin, “High-speed nonlinear finite element analysis for surgical simulation using graphics processing units,” *Medical Imaging, IEEE Transactions on*, vol. 27, no. 5, pp. 650–663, May 2008.
[15] ParMETIS, http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.
[16] PETSc, http://www.mcs.anl.gov/petsc/petsc-as/.
[17] Gmm++, http://home.gna.org/getfem/gmm_intro.