Parallel Mutual Information Based 3D Non-Rigid Registration on a Multi-Core Platform

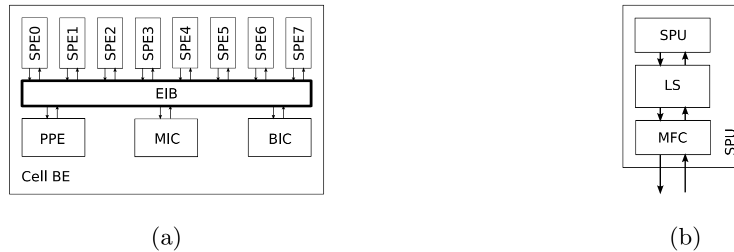
Jonathan Rohrer[1], Leiguang Gong[2], and Gábor Székely[3]

[1] IBM Zurich Research Laboratory, 8803 Rüschlikon, Switzerland
jon@zurich.ibm.com,
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
[3] Computer Vision Laboratory, ETH Zurich, 8092 Zurich, Switzerland

Abstract. For many clinical applications, non-rigid registration of medical images demands cost-effective high performance solutions in order to be usable. We present a parallel design and implementation of a well-known algorithm based on mutual information and a B-Spline transformation model for the Cell Broadband Engine™(Cell/B.E.) platform. It exploits the different levels of parallelism offered by this heterogeneous multi-core processor and scales almost linearly on a system containing two chips or a total of 16 accelerator cores. A speedup of more than 40x was reached compared to a sequential implementation and registration time for a 512x512x100 voxel image was 60 seconds.

1 Introduction

Image registration is the process of finding a dense transformation between two images, for example to fuse images acquired with different devices or at a different time or to compensate anatomical differences between images of different subjects. It has numerous applications in medical computing and different algorithms exist. To be able to deal with soft tissue, non-rigid registration is needed, which is a computationally highly demanding task, therefore it has been very slowly moving into routine applications in real clinical environments. The effort has been intensified in recent years to develop viable approaches for ever faster registration implementations on various types of hardware platforms and configurations. While research continues to explore strategies for more efficient and rapidly converging methods, increasing attention has been given to parallelization of algorithms for emerging high performance architectures, such as cluster and grid computing, as well as advanced graphical processing units (GPU) and high power multi-core computers. In [1] a supercomputer-based parallel implementation of multimodal registration is reported for image guided neurosurgery using distributed and grid computing. Another supercomputer-based approach is presented in [2]. One of the practical issues with these architectures is availability and cost, since even for mid-size clinics financial limitations can be a major problem.



(a) (b)

Fig. 1. (a) The Cell/B.E. processor is a multi-core processor with one PPE and eight SPEs. (b) An SPE consists of a synergistic processing unit (SPU), a 256 KB local store (LS) and a memory flow controller (MFC).

The recent emergence of low cost, high power multi-core processor systems has opened up an alternative venue for developing cost-effective high-performance medical imaging techniques based on parallel processing. In this paper, we report on an implementation of a mutual-information-based non-rigid registration algorithm on the Cell/B.E. platform to investigate the performance gain potential. We explored how to partition the registration problem into parallel tasks that can be mapped onto the heterogeneous Cell/B.E. platform with its general purpose processor and attached accelerator cores. We implemented a sequential version of a well-known non-rigid registration algorithm based on free-form deformations using B-Spline interpolation. We then parallelized the same registration method for the Cell/B.E. platform. Experiments with different datasets were carried out. Running on a two processor system, the performance analysis shows a speedup of more than 40x compared to the sequential version.

2 Cell/B.E. architecture

Recently, the processor industry is moving towards chip multiprocessors due to diminishing performance returns of single processor designs on chip area and power. The first-generation Cell/B.E. processor is a heterogeneous chip multiprocessor combining a Power processor element (PPE) with eight synergistic processor elements (SPEs) [3] (figure 1a). The SPEs are accelerator cores specialized on single precision floating point calculations that account for the majority of the compute performance. An SPE consists of two units, the SPU (synergistic processing unit) and the MFC (memory flow controller) (figure 1b). An SPE operates on its 256 KB LS (local store), which contains instructions and data. The MFC transfers data blocks between the local store and system memory. These transfers are initiated by DMA (direct memory access) commands, meaning that the programmer has to explicitly move data into the local store to make it available to the SPU and then explicitly transfer results back to main memory. The SPU is based on a 128bit SIMD architecture (single instruction multiple data), allowing for example to execute four single precision float operations in parallel

as one vector operation. The SPEs and the PPE are communicate through the EIB (element interconnect bus).

The Cell/B.E. processor can outperform other competing processors by around an order of magnitude for a variety of workloads [4]. Mutual information based rigid registration has been accelerated by a factor of 11x on a dual Cell/B.E. system [5]. The performance benefits stem from various forms of parallelism offered by the architecture [6], for example data-level parallelism in the SPU with its SIMD instruction support, parallel DMA transfers and computation in the SPE and thread-level parallelism because of the multi-core design. A common approach to programming for the Cell/B.E. processor is to have a main thread running on the PPE orchestrating the threads on the SPEs, to which the majority of the computational workload is offloaded.

3 Parallelization of the registration algorithm

3.1 MI based non-rigid registration algorithm

The registration algorithm we implemented models the transformation function based on B-Splines, which is a well-known approach to non-rigid registration [7] [8]. The idea is that a mesh of control-points is manipulated and the dense transformation function $\mathcal{T}(\mathbf{x};\boldsymbol{\mu})$ is obtained by B-Spline interpolation. The finite set of parameters, $\boldsymbol{\mu}$, is the set of B-Spline coefficients located at the control points. The degrees of freedom of the model can be adjusted through the control-point spacing. To register a floating image I_{flt} to a fixed image I_{fix}, the parameters $\boldsymbol{\mu}$ are optimized iteratively with the goal to find the best matching. A gradient descent optimizer with feedback step adjustment is used [9]. In order to support the registration of images of different modalities, Mutual Information [10] [11] is used as the similarity metric to be optimized. Its calculation is based on a Parzen estimation of the joint histogram $p(i_{fix}, i_{flt})$ of the fixed and the transformed floating image [7]. As proposed in [12], a zero-order spline Parzen window for the fixed image and a cubic spline Parzen window for the floating image is used. Together with a continuous cubic B-Spline representation of the floating image, this allows to calculate the gradient of the metric S in closed form. The contribution of the point at coordinates $\mathbf{x}$ in the fixed image to the derivative of S with respect to the transformation parameter μ_i is:

$$\begin{aligned} \frac{\partial S}{\partial \mu_i} = & -\alpha \left. \frac{\partial p(i_{fix}, i_{flt})}{\partial i_{flt}} \right|_{i_{fix}=I_{fix}(\mathbf{x}),\, i_{flt}=I_{flt}(\mathcal{T}(\mathbf{x};\boldsymbol{\mu}))} \\ & \cdot \left(\frac{\partial}{\partial \boldsymbol{\xi}} I_{flt}(\boldsymbol{\xi})\big|_{\boldsymbol{\xi}=\mathcal{T}(\mathbf{x};\boldsymbol{\mu})} \right)^{\mathsf{T}} \cdot \frac{\partial}{\partial \mu_i} \mathcal{T}(\mathbf{x};\boldsymbol{\mu}), \end{aligned} \tag{1}$$

where α is a normalization scale factor. The high-order interpolation of the floating image is justified in spite of its higher computational complexity because of its advantages in terms of registration accuracy but also convergence speed because of the better estimations of the image gradient [13].

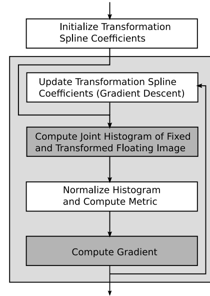


Fig. 2. The algorithm iteratively optimizes the transformation coefficients

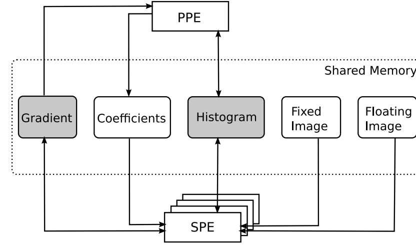


Fig. 3. The PPE thread and the SPE threads share several data arrays that are in main memory

We use all the voxels of the fixed image to calculate the histogram and not only a subset like in [12]. But we use a multi-resolution approach to increase robustness and speed [7], meaning that we first register with reduced image and transformation resolutions, and then successively increase them until the desired resolution has been reached.

3.2 Serial implementation

We used a serial implementation of the algorithm to analyze the performance and identify candidate functions for parallelization. The registration of the floating to the fixed image is an iterative optimization of the transformation coefficients after they were initialized (figure 2). Initialization can for example base on a previously solved coarser level of the multi-resolution approach, centric alignment of the two images or manual alignment. We can divide an iteration into several sub-steps. In a first step, we have to calculate the joint histogram of the fixed and the transformed floating image. For every point in the fixed image, the corresponding point in the floating image is found (based on the current transformation function parameters), and the floating image is evaluated at these coordinates. An entry for the point pair is added in the joint histogram. We do not store the transformed floating image, although it might save processing time later. It would be especially beneficial if we also would evaluate the gradient of the transformed floating image already. We decided to not do so in order to keep memory consumption low, which may be important when processing large datasets.

Second, the histogram has to be normalized to obtain the joint pdf (probability density function) and the marginal pdfs, which allows to calculate the mutual information of the two images. In the third step, the gradient is calculated, based on the sum of the contributions of all the point pairs. This gradient is used in the fourth step by the gradient descent optimizer to calculate the new set of transformation parameters. An iteration is only successful if the mutual information of the two images has increased. If so, the step size of the optimizer

is increased for the next iteration. If not, we discard the new set of parameters and decrease the step size.

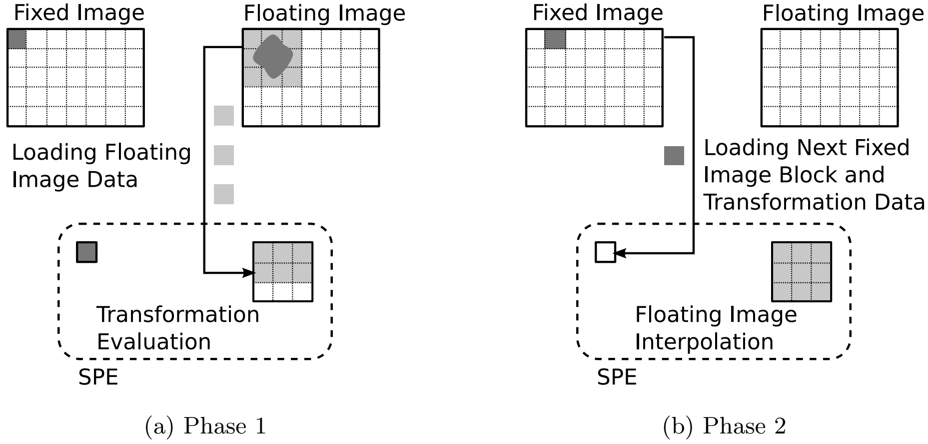
If a step is unsuccessful, calculation of the gradient would not be necessary. We calculated the gradient in every step in order to make the runtime less dependent on the ratio of successful and unsuccessful steps. For the data in section 4.2, calculation of the joint histogram and the gradient (steps 1 and 3) together account for 99.9967% of the optimization runtime, making them best candidates for parallelization.

3.3 Thread Level Parallelism

Based on the characteristics of the hardware platform and the performance analysis of the single threaded implementation, our approach to speed up the algorithm was to offload the calculation of the joint histogram and the gradient to the SPEs, while the rest of the algorithm, like histogram normalization and coefficient update by the gradient descent optimizer is still running sequentially on the PPE. If we want to use n SPEs for the calculations, we can partition the fixed image into n subsets and assign each one to an SPE. Each SPE only calculates the contribution of its part to the joint histogram or the gradient respectively and in the end the partial results are summed up. The SPE threads are invoked before starting with the iterations and are waiting for commands from the main thread running on the PPE. When the algorithm has to calculate the joint histogram, it sends a message to each SPE and waits until it receives a completion message from all SPEs, which they send out upon finishing. Similar synchronizing communication is used for the calculation of the gradient. Having one program capable of performing all the necessary computations running all the time on the SPEs saves us the overhead of spawning the SPE threads for every function call.

The PPE thread and the SPE threads share several data arrays that are in main memory (figure 3), which may make it necessary to use synchronization mechanisms if several threads possibly want to write to the same location and also can lead to coherency problems in the explicitly administered cache in the local store of the SPE. The fixed and floating image data is never modified throughout the optimization process, therefore there is no danger incoherency and no need for synchronization. The coefficients of the transformation function are only modified by the PPE thread (during gradient descent optimization), so we again need no synchronization here. To avoid coherency problems, we simply invalidate all the cached coefficient data in the LS at the beginning of every iteration.

During computation, all the SPEs have to write to the joint histogram or the gradient. These arrays are relatively small compared to the image arrays. To avoid synchronization overhead, we keep for each SPE a separate copy in main memory on which it can work. After they finish, the PPE thread gathers the partial results.



(a) Phase 1

(b) Phase 2

Fig. 4. Processing a fixed image block consists of two phases, allowing to overlap calculations with data transfers.

3.4 Hiding memory latency

Because an SPE can only keep a limited amount of data in the LS, we have to dynamically fetch the required data from main memory by DMA commands. DMA is especially efficient if large, continuous blocks of data can be transferred at once, therefore we partition the fixed and the floating image into cuboids and each of them is mapped to a continuous memory region [14]. If we can pre-fetch data to the LS before it is needed, we can keep the processing unit busy and avoid stalls caused by the latency of main memory, which is much higher than the latency of the LS. We achieve this by processing one block of the fixed image at a time. We first fetch the reference image block and the coefficients that are necessary to evaluate the transformation function in the region of this block. Then, while we calculate for each point $\mathbf{p}$ in the fixed image the coordinates $\mathbf{p}' = \mathcal{T}(\mathbf{p})$ it is mapped to in the floating image space, we start fetching the floating image blocks that overlap with the transformed fixed image block (figure 4a). By the time we have evaluated the transformation, the floating image data is available. We can then interpolate the image at the coordinates $\mathbf{p}'$ and add an entry to the joint histogram for the detected intensity pair. While we do these calculations, the next fixed image block can already be pre-fetched together with the coefficients relevant for the evaluation of the field for the points in the block (figure 4b).

3.5 SPE local store organization

We use a software managed cache to organize the floating image data in the local store. The cache has 27 (3x3x3) slots with a direct image block to cache

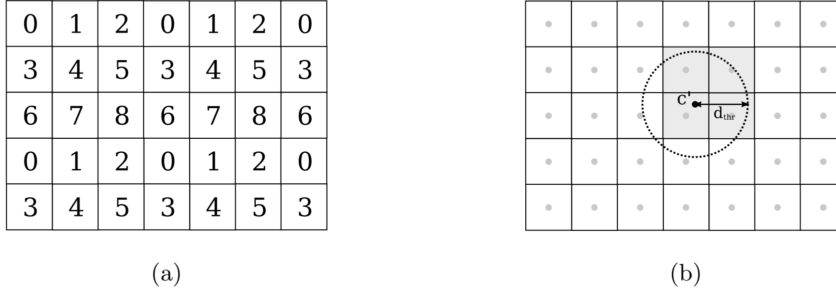


Fig. 5. The caching schemes are shown for 2 dimensions for ease of visualization. (a) Every block of the floating image has a fixed mapping to a cache slot in the LS. (b) Blocks are only fetched if the distance between their center and the mapped center of the reference block c' is below the threshold d_{thr}

block mapping (figure 5a). The cache index for a block $\bar{\mathrm{B}}_{i_x,i_y,i_z}$ is calculated as

$$(i_x \text{ MOD } 3) + (i_y \text{ MOD } 3) \times 3 + (i_z \text{ MOD } 3) \times 9. \quad (2)$$

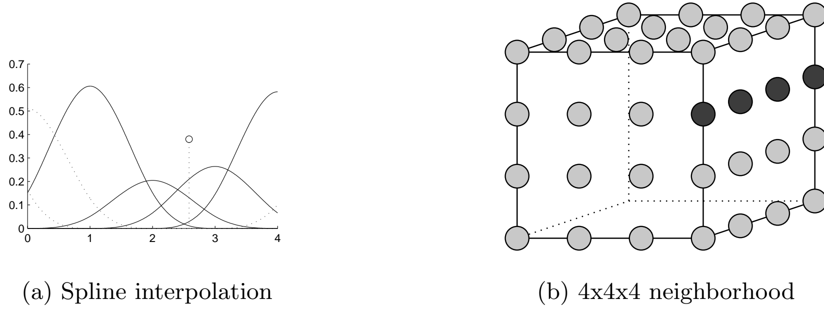
To decide which blocks of the floating image to fetch, we calculate the mapping of the point in the center of the fixed image block $\mathbf{c}' = \mathcal{T}(\mathbf{c})$. The block containing $\mathbf{c}'$ and its 26 neighbors are loaded to the LS, which should cover the entire mapped fixed image block assuming that the voxel size in both images is the same and $\mathcal{T}$ does not stretch extremely into any particular direction. If we process neighboring fixed image blocks sequentially, chances are high that a large part of the required floating image blocks were already fetched while processing the preceding neighbor and we do not have to re-fetch them.

With this approach it is probable that we fetch blocks that are not needed and that we unnecessarily stress the memory bandwidth. We introduce therefore the parameter d_{thr} and only fetch blocks $\bar{\mathrm{B}}_{i_x,i_y,i_z}$ whose center point $\bar{\mathbf{c}}_{i_x,i_y,i_z}$ meets the condition $\left|\left|\bar{\mathbf{c}}_{i_x,i_y,i_z} - \mathbf{c}'\right|\right| < d_{thr}$ (figure 5b).

In the case that for a $\mathbf{p}'$ we do not have the corresponding image data in the LS, we skip it. Instead of fetching the missing image data, we send an exception message to the PPE thread that it has to process the point-pair $\mathbf{p} - \mathbf{p}'$. In order not to have to send a message for every cache miss, we can accumulate up to 64 and send them together.

3.6 SIMD implementation

A large part of the computation time stems from the evaluation of the transformation function $\mathbf{p}' = \mathcal{T}(\mathbf{p})$ and the interpolation of the floating image at the mapped coordinates $I_{flt}(\mathbf{p}')$. Both base on cubic B-Spline interpolation: $\beta_3(\mathbf{x}) = \prod_{k=1}^{N} \beta_3(x_k)$, with $\mathbf{x} = (x_1, ..., x_N)$ and $N = 3$ for the 3 dimensional case. I.e.$\beta_3(\mathbf{x})$ is a tensor product of cubic B-Splines.



(a) Spline interpolation (b) 4x4x4 neighborhood

Fig. 6. (a) In the 1D case of cubic spline interpolation, only 4 splines have to be evaluated to compute the interpolated value at a given position. The splines that can be ignored to calculate the value marked with a circle are dashed. (b) In the 3D case, interpolation works on 4x4x4 blocks of data, 4 elements can be processed at a time in the SIMD unit.

$$\mathcal{T}\left(\mathbf{x}\right) = \mathbf{x} + \sum_{\mathbf{j}} \mathbf{c}_{\mathbf{j}} \beta_3 \left(\frac{\mathbf{x}}{\mathbf{h}} - \mathbf{j}\right) \tag{3}$$

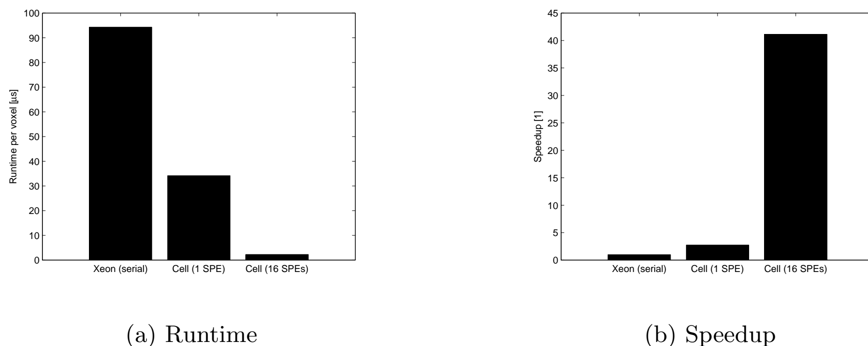
where $\mathbf{j}$ are the parameter indices and $\mathbf{h}$ is the mesh spacing.

$$I_{flt}\left(\mathbf{x}\right) = \sum_{\mathbf{i}} b_{\mathbf{i}} \beta_3 \left(\mathbf{x} - \mathbf{i}\right) \tag{4}$$

Because of the limited support of the cubic B-Splines, the evaluation in the 1D case is a weighted sum (weight $\beta_3\left(x\right)$) of 4 coefficients (figure 6a). In the 3D case we therefore have to work on a 4x4x4 neighborhood of $\mathbf{x}$ to evaluate $\mathcal{T}\left(\mathbf{x}\right)$ or $I_{flt}\left(\mathbf{x}\right)$.

The neighborhood size matches well with the 4-way SIMD unit of the SPU. A requirement to efficiently use the SIMD unit is that the vectors that are processed are stored continuously in memory. If the elements have to be loaded first individually to assemble the vectors, the performance benefit is lost. When storing 3D data to memory, adjacent elements along one coordinate direction can be chosen to be neighbors in memory, like the highlighted 4 elements in (figure 6b), so we can meet this requirement.

Performance is also decreased if the start of the vector data is not aligned to a 16 byte address boundary in memory, because the loading has to be split into two steps. We need to load vectors starting at any data element, therefore we can not directly meet this requirement. We introduce the constraint that the spline mesh spacing is a multiple of the image block size. Like this, the same coefficients are needed to evaluate the transformation function for all the points within an image block. They are loaded and aligned before processing of the block starts.



(a) Runtime (b) Speedup

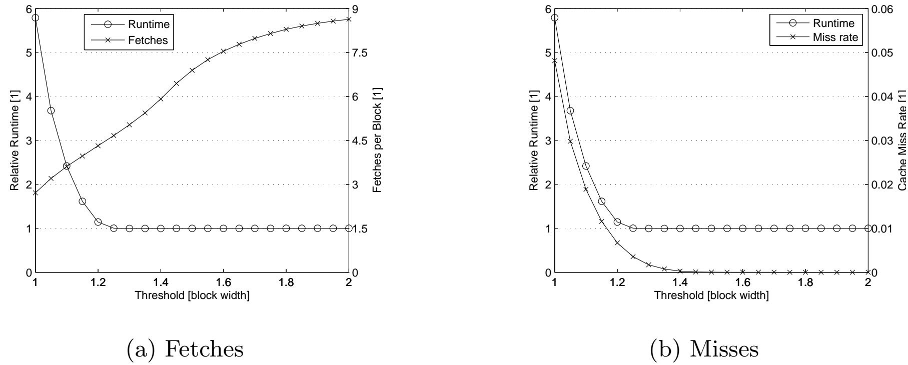
Fig. 7. Absolute algorithm runtime and speedup

4 Experimental results

For the performance measurements, two different systems were used. The parallel version was running on an IBM® BladeCenter® QS20 featuring 1 GB of RAM and two processors running at 3.2 GHz configured as a two-way, symmetric multiprocessor (SMP) [15]. A thread running on a PPE can communicate with all the 16 SPEs. The Cell/B.E. SDK 3.0 and the GCC compiler were used to implement and compile the algorithm. In all the experiments, there was one main thread running on one of the PPEs and up to 16 threads on the SPEs. The sequential version was running on one core of an Intel® Xeon™3.2GHz with 4 GB of RAM. The code was compiled with the GCC compiler with the flags set to use sse2, but the code was not tuned for the SIMD unit. On both systems, we measured only the runtime of the iterative optimization (box in figure 2), assuming that all the data was already loaded to main memory the way the algorithm expects it.

4.1 Runtime

In this experiment, we carried out 60 registrations of CT abdominal images with different sizes and measured the mean of the registration time per voxel. A four level multi-resolution pyramid was used with a B-Spline grid width of 16x16x16 voxels in the finest level. The gradient descent optimizer was set to carry out a fixed number of 30 iterations at every level. The sequential version required 94.3 μs/voxel (about 41 minutes for an image size of 512x512x100 voxels) (figure 7a). The Cell/B.E. version using 1 SPE completed in 34.2 μs/voxel (15 minutes for the same image). This speedup of 2.75x (figure 7b) stems from the restructuring of the code for SIMDization and parallelism of data transfers and calculation. The threshold d_{thr} was set to 2 block widths, which was high enough so that no cache misses occured. Therefore the PPE thread was busy waiting while the SPE thread was calculating and the two threads never worked in parallel.



(a) Fetches (b) Misses

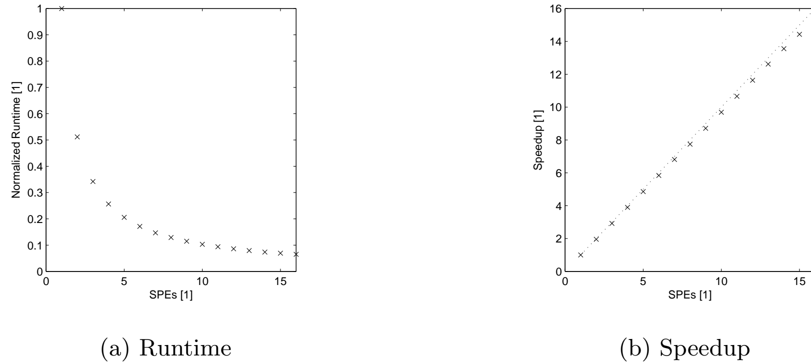
Fig. 8. Runtime (relative to the runtime for $d_{thr} = 2$), floating image block fetches per fixed image block and cache misses for different values of d_{thr} (in units of floating image block width)

When exploiting the thread level parallelism by using all the 16 SPEs, the runtime was further lowered to 2.29 μs/voxel (1 minute for the image of size 512x512x100 voxels). The overall speedup compared to the sequential implementation was 41.1x.

4.2 Memory traffic

In this experiment we used all the 16 SPEs on the QS20 and looked at the effect of d_{thr} on the algorithm runtime. An MR image from the Brainweb Project with a size of 181x217x181 was used. It was deformed by random transformations and then registered back to the original image. We calculated the mean values based on registrations with 50 different random transformations. The spline grid spacing was 8 voxel units (8 mm).

The effect of a lower threshold is that less floating image blocks are fetched, leading to less memory traffic. Figure 8 shows that for $d_{thr} = 2$ (in units of floating image block widths) almost 9 blocks have to be fetched per fixed image block. This number decreases when lowering d_{thr} and for $d_{thr} = 1.25$ we only have to fetch around 3 blocks. The algorithm runtime remains unaffected in this range, indicating that the algorithm is not memory bound. Below 1.25 a further decrease of d_{thr} causes the runtime to rise rapidly and for $d_{thr} = 1$ it is almost six times larger than for $d_{thr} = 2$. In figure 8b we see that for a threshold below 1.4 we start to have cache misses, meaning that not all the required floating image data is in the LS of the SPEs and more and more calculation has to be done in the PPE. For $d_{thr} < 1.25$ the processing of the cache misses in the PPE starts to take longer than the normal processing in the SPEs and becomes the determining factor for the overall runtime. This is already the case for a cache miss rate of less than 0.5% because the cache miss processing in the PPE



(a) Runtime

(b) Speedup

Fig. 9. Scalability

is much less efficient than the processing in the SPEs and requires additional communication.

4.3 Scalability

In this experiment we performed 10 registrations with the same images as in section 4.1 with the same settings and measured the mean runtime depending on the number of SPEs used (figure 9a, plotted is the relative runtime compared to utilization of 1 SPE). In figure 9b we see that the algorithm scales almost perfectly. The SPE efficiency for 16 SPEs ($\frac{t_1}{16 \cdot t_{16}}$, where t_n is the runtime for n SPEs) is 0.958.

5 Discussion

High computational cost is a main obstacle for the usability of non-rigid registration in a clinical environment. The work presented in this paper shows that this task can be accelerated significantly on the Cell/B.E. processor. The algorithm was restructured to exploit the different forms of parallelism the architecture offers, which allowed to achieve good scalability. The execution time was lowered to 2.29μs/voxel on a system with two processors. An implementation of mutual information based non-rigid registration on a supercomputer using 64 CPUs was reported to reach a speedup of up to 40x compared to 1 CPU, resulting in a mean execution time of 18.05μs/voxel for that algorithm [2]. Even if one has to be careful when comparing these numbers because the implemented algorithms differ and runtime may depend on several parameters (like control-point spacing or number of iterations), our results demonstrate that the Cell/B.E. architecture can achieve comparable or better performance with much less processor chips. It even outperforms a hardware implementation presented in [16], which is reported to have a runtime of roughly 3.6μs/voxel.

References

1. Chrisochoides, N., Fedorov, A., Kot, A., Archip, N., Black, P., Clatz, O., Golby, A., Kikinis, R., Warfield, S.: Toward real-time, image guided neurosurgery using distributed and grid computing. In: SuperComputing06, Tampa, Florida, USA (2006)
2. Rohlfing, T., Maurer Jr., C.R.: Nonrigid image registration in shared-memory multiprocessor environments with application to brains, breasts, and bees. IEEE Trans. Inf. Technol. Biomed. **7**(1) (2003) 16–25
3. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. **49**(4/5) (2005) 589–604
4. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation - a performance view. IBM J. Res. Dev. **51**(5) (2007) 559–572
5. Ohara, M., Yeo, H., Savino, F., Gong, L., Inoue, H., Komatsu, H., Sheinin, V., Daijavad, S., Erickson, B.: Real-time mutual-information-based linear registration on the Cell Broadband Engine processor. In: Proc. ISBI. (2007) 33–36
6. Gschwind, M.: The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. International Journal of Parallel Programming **35**(3) (2007) 233–262
7. Thévenaz, P., Unser, M.: Spline pyramids for inter-modal image registration using mutual information. In: Proc. SPIE. Volume 3169. (1997) 236–247
8. Rueckert, D., Sonoda, L., Hayes, C., Hill, D., Leach, M., Hawkes, D.: Nonrigid registration using free-form deformations: Application to breast MR images. IEEE Trans. Med. Imag. **18**(8) (1999) 712–721
9. Kybic, J., Unser, M.: Fast parametric elastic image registration. IEEE Trans. Image Process. **12**(11) (2003) 1427–1442
10. Viola, P., Wells III, W.M.: Alignment by maximization of mutual information. Int. J. Comput. Vision **24**(2) (1997) 137–154
11. Maes, F., Collignong, A., Vandermeulen, D., Marchal, G., Suetens, P.: Multimodality image registration by maximization of mutual information. IEEE Trans. Med. Imag. **16**(2) (1997) 187–198
12. Mattes, D., Haynor, D.R., Vesselle, H., Lewellen, T.K., Eubank, W.: PET-CT image registration in the chest using free-form deformations. IEEE Trans. Med. Imag. **22**(1) (2003) 120–128
13. Kybic, J., Thévenaz, P., Nirkko, A., Unser, M.: Unwarping of unidirectionally distorted EPI images. IEEE Trans. Med. Imag. **19**(2) (2000) 80–93
14. Jianchun, L., Papachristou, C.A., Shekhar, R.: A "brick" caching scheme for 3d medical imaging. In: ISBI, IEEE (2004) 563–566
15. Nanda, A.K., Moulic, J.R., Hanson, R.E., Goldrian, G., Day, M.N., D'Amora, B.D., Kesavarapu, S.: Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. IBM J. Res. Dev. **51**(5) (2007) 573–582
16. Dandekar, O., Wahmbe, V., Shekhar, R.: Hardware implementation of hierarchical volume subdivision-based elastic registration. In: Proc. EMBS. (2006) 1425–1428

BladeCenter and IBM are trademarks of IBM Corporation. Intel is a trademark of Intel Corporation. Intel Xeon is a trademark of Intel Corporation. Cell Broadband Engine is a trademark of Sony Computer Entertainment Inc. Other company, product, or service names may be trademarks or service marks of others.