10.1109/ULTSYM.2009.0340

Performance optimization of block matching in 3D echocardiography

Gabriel Kiss*, Espen Nielsen†, Fredrik Orderud† and Hans G. Torp*
*Norwegian University of Science and Technology, Department of Circulation and Medical Imaging, Trondheim, Norway
†Norwegian University of Science and Technology, Computer Science Department, Trondheim, Norway

***Abstract*—The introduction of real time 3D echocardiography allows for deformation tracking in three dimensions, without the limitations of 2D methods. However, the processing needs of 3D methods are much higher. The aim of the study was to optimize the performance of 3D block matching by using a Single Instruction Multiple Data (SIMD) model, a technique employed to achieve data level parallelism. Two implementations of SIMD have been tested. The first is based on Streaming SIMD Extensions (SSE), the second uses CUDA, a SIMD architecture proposed by NVIDIA, which is available on several graphics cards. The proposed methods have been validated on synthetic and patient data. With the use of SIMD architecture, the overall processing time is significantly reduced, thus making 3D speckle tracking feasible in a clinical setting. Apart from the implemented sum of square differences (SAD), other matching criteria (e.g. sum of squared differences, normalized cross correlation) can be implemented efficiently (especially on the GPU) thus further improving the accuracy of the block matching process.**

I. Introduction

Speckle tracking echocardiography [1], [2] is a non-invasive method for assessing left ventricular function. 2D speckle tracking has already become a clinically validated tool for both global and regional LV function. It is able to estimate 2D strain components in the imaging plane [3], by determining the movement of the interference pattern caused by subresolution scatteres, an inherent characteristic for both 2D and 3D US acquisition methods.

With the introduction of real time 3D echocardiography, the full deformation tensor can be estimated. This eliminates the limitations of 2D methods and 3D motion tracking, similar to cardiac magnetic resonance imaging [4] is achievable. With the added benefit of high temporal resolution, 3D echocardiography could become a prime candidate for clinical cardiac motion analysis.

Several methods for 3D speckle tracking both on simulated and phantom data, as well as on patient data have been proposed. Early simulation work by Meunier [5] was followed by Morsy et al. [6]. Orderud et al. [7] proved on phantom data that a segmentation approach can be extended by speckle tracking in order to estimate segmental strain values. Song [8] used motion coherence of septal points for speckle tracking. Crosby et al. [9] developed a speckle tracking method robust enough to be used on clinical recordings. Alternatively, Elen et al. [10] proposed a method based on spatio-temporal elastic registration, in which image registration is used to estimate the 3D strain tensor.

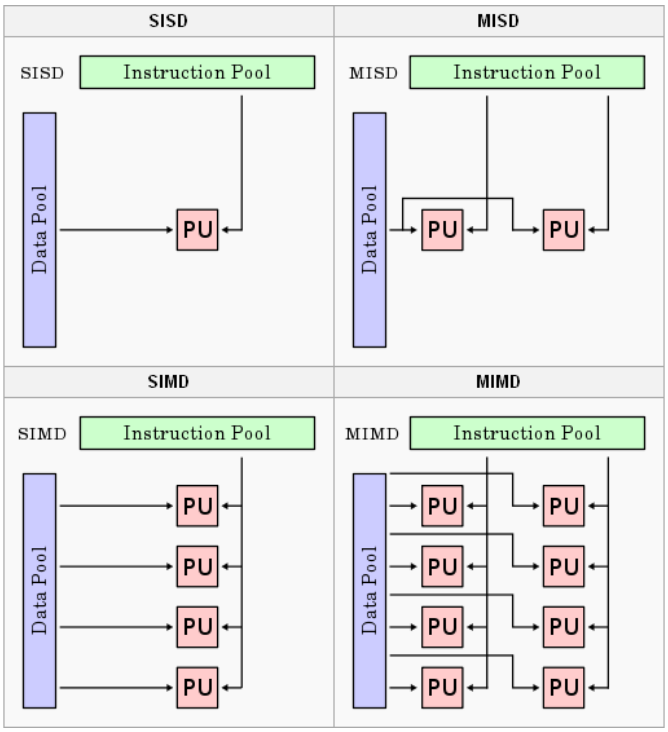

Figure 1: Computer architecture classification as proposed by Flynn. Given a number of processing units (PU), single (S) or multiple (M) instruction sets (I) and data streams (D) can be processed concurrently. (source Wikipedia)

However, the computational requirements for 3D speckle tracking are highly increased when compared to the 2D case. Even at 20-30 frames per cardiac cycle the speckle tracking takes from around 4 minutes [9] up to 10-20 minutes [10] to complete. In the case of speckle tracking approaches, most of the time consists of block-matching operations, i.e. searching the optimal displacement that minimizes or maximizes a given matching criteria between two frames.

II. Methods

The purpose of this study was to improve the computational efficiency of the block matching process. Crosby et al. [9] estimate the local deformation field by tracking and regularizing the motion of the LV represented as a quadrilateral mesh.

Given an initial mesh, speckle tracking is performed at each material point (i.e. vertex) by block-matching, with

SAD as the matching criteria. The block-matching process is repeated for all the points in the mesh, typically each mesh consists of approximately 600 vertices. In a second step the integer displacement values are refined to achieve sub-voxel accuracy. Points that yield poor tracking results are discarded and missing values are filled in by using a spring metaphor. To further increase the tracking accuracy, both forward and backward tracking is carried out and the final deformation vector is obtained by linear weighting of the forward and backward results.

For comparison purposes a reference block matching algorithm, as illustrated in figure 2, was implemented. Its pseudocode is listed below.

```
foreach tracking_point {
  foreach search_offset {
    metric = 0
    foreach kernel_offset {
      metric += compute metric (
        kernel_volume(kernel_offset),
        search_volume(search_offset
                    + kernel_offset)
                )
    }

    if compare(metric, optimal_metric ) {
      optimal_metric = metric
      optimal_offset = search_offset
    }
  }
}
```

Conceptually the tracking process can be parallelized at 2 different levels and can be represented by a Single Instruction Multiple Data (SIMD) model, at each of these levels, according to the definition of M.J. Flynn [11]. First the tracking of all the mesh points can be done in parallel, second the computation of the matching values per mesh point, can be parallelized as well.

In contrast to the traditional method for computer architecture, that has a single processing unit with a single instruction set that computes one instruction at a time, a SIMD architecture can execute the same set of instructions simultaneously on multiple data blocks (Figure 1). The SIMD model was first implemented on large supercomputers, however most of today's CPUs and GPUs have support for SIMD instructions of some degree, although the GPU can be seen as a MIMD device as well. In the following subsections the optimized block matching algorithms are presented.

A. Block matching on the CPU - Streaming SIMD Extensions (SSE)

Streaming SIMD Extensions (SSE) is an extension developed by Intel for their x86 architecture. In addition to the standard 32 bit registers, x86 computers are equipped with separate 128 bit registers, which are accessible to the SSE instructions (an additional set of approximately 70 instructions). SSE enables processing of these registers as if they

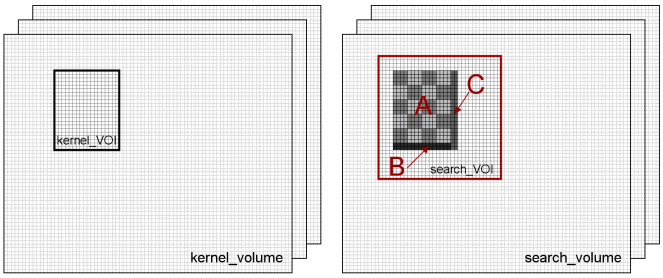

Figure 2: The motion vector of a material point, between a 'kernel_volume' and a 'search_volume', is found as the displacement of a region of interest around the point 'kernel_VOI' that results in the best match within a 'search_VOI'. Several matching metrics have been proposed, the most common include: sum of absolute differences (SAD), sum of squared differences (SSD) or normalized cross-correlation.

contained an array of values, instead of a single one. For integer operations, SSE instructions can regard the registers as containing: one 128 bit value, two 64 bit values, four 32 bit values, eight 16 bit values or sixteen 8 bit values. The latter representation is the most convenient for US data which consists of 8 bit gray scale voxel volumes for each frame.

The fast block difference instruction $\_mm\_sad\_epu8$ can compute the SAD value for sixteen 8 bit values by using only 9 CPU instructions [12]. It is a small scale SIMD model, however it can lead to a theoretical 16x increase in the processing speed. In practice, the transfer overhead to and from the 128 bit registers decreases the actual speed-up factor.

Data alignment is the most important factor for optimal results. It would be preferable to have a 16 byte alignment of the data, but that is not always the case due to varying sizes of the kernel and search blocks. In order to handle all possible block sizes, the following technique for computing the SAD values was adopted:

1) Each 3D kernel and search sub-volumes of size (#row, #column, #plane) voxels are decomposed into #plane planes that are processed independently.
2) As much as possible of the plane is partitioned into squares of 4x4 8-bit elements (area A in figure 2). Each of these 4x4 squares is transferred to the 128 bit registers by using only 4 read operations $\_mm\_set\_epi32$. For the current kernel and search offsets, the SAD value in each of the 4x4 squares is computed by one operation $\_mm\_sad\_epu8$.
3) The end of the columns (area B in figure 2) are processed similar to area A. The difference here is that extra rows of zeros are added to form squares of 4x4 elements. These extra rows have to always be matched against zeroes instead of values from the search window. An exception from this row padding is when area B only consists of one row. Then all the elements are processed as single elements without SIMD optimization.
4) The end of the rows are processed as single elements without SIMD optimization (area C in figure 2).
5) Contributions belonging to areas A, B and C are added together using $\_mm\_add\_epi64$, to obtain the block's

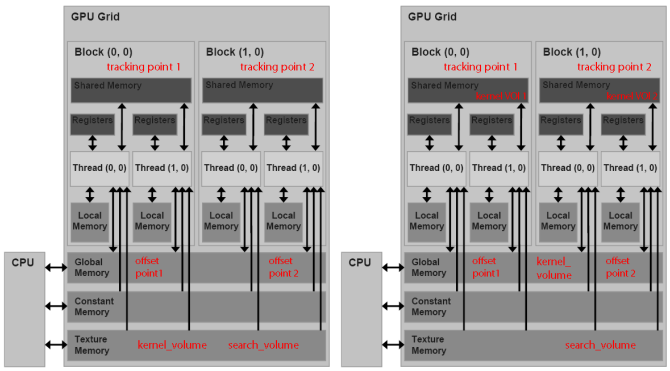

Figure 3: Two possible memory allocation approaches for the CUDA implementation. (a) Both the kernel and search volumes are stored in the texture memory. (b) Since for a tracking point the kernel_VOI is the same, for all search_offsets, the kernel volume is placed into global memory and the kernel_VOI is transferred to shared memory. The search_volume is stored into texture memory.

SAD value.

B. Block matching on the GPU - CUDA

Traditionally, GPUs were specifically optimized for graphics (e.g. texture mapping and polygon rendering) and were very restrictive in terms of operations and programming. However, as the GPUs were developing into multi-core processor units, the main GPU designers (i.e. NVIDIA and ATI) became aware of their usefulness as general purpose computing devices.

In 2006, NVIDIA introduced a modified form of their standard stream processor which allowed general purpose computations on the GPU. This required both hardware changes (most notably the addition of the thread execution manager), a redesigned software stack and a new programming model. CUDA ("Compute Unified Device Architecture") extends the standard C language by adding a set of instructions that allow programmers to define CUDA functions called 'kernels' which are executed in parallel as CUDA 'threads', manage the GPU memory and synchronize among different threads. To increase scalability, the threads are subdivided into multiple 'blocks'. Only threads within the same block can cooperate via shared memory and synchronization calls. Thus, CUDA allows CPU-based applications to access directly the resources of a GPU and eliminates the limitations of using a graphics API. The CUDA architecture can be seen as a SIMD model since the same instruction set, defined by the 'kernel', is executed in parallel by 'threads' on different data.

Several approaches can be taken when implementing the block matching algorithm on the GPU. First the allocation of CUDA blocks and threads has to be chosen, second one has to decide where to store the image data. Since the computation of the SAD values at different locations is an independent process, each material point is assigned to a different block. Because the maximum number of threads per block is limited, instead of computing one SAD value per thread, each thread is extended to compute several SAD values (total number of SAD computations per material point divided by the maximum number of threads per block, which is a user defined constant). The SAD value is computed using the $\_\_usad(x, y, z)$ instruction, which is a GPU optimized instruction that returns $|x - y| + z$. After completion each block returns the minimum SAD value and its position in the search volume.

Two different memory allocation schemes have been tested, one represents both the kernel and the search volumes as textures, while the second uses texture for the search volume while the kernel data is stored in the GPU's global memory. Because the kernel data is the same for all search offsets, each block will load it into a very low latency shared memory, to further optimize the process. Figure 3 presents the allocation of resources for both cases.

III. Experiments and Results

The previously described methods were implemented in C++, CUDA and validated on synthetic and 3D US patient data. Test systems included a Dell Latitude 830 laptop: NVIDIA Quadro NVS 135M, 6.4GB/s bandwidth, Intel Core2 Duo T7500, 2GB RAM, 3112 SSE MFLOPS/register and a high-end graphics system: NVIDIA GeForce GTX 285, 158 GB/s bandwidth, Intel Core i7 920 Quad Core Processor, 3GB RAM, 3742 SSE MFLOPS/register.

The synthetic data consisted of 200x200x40 voxels, kernel VOI of 13x13x9 voxels and a search VOI of 25x25x17 voxels. The number of tracking points was varied from 1 to 1000, for each method the average computation time over 10 runs was recorded and is presented in figure 4 (a), (d). The speed-up factors are shown in figure 4 (b), (e).

Prior work on 2D block matching can also be optimized by incorporating the presented methods. Initial results are presented in figure 4 (c), (f), using synthetic data 200x200 pixels, kernel ROI 11x11 pixels and search ROI 21x21 pixels.

On the desktop computer, CUDA initialization and data transfer to the GPU memory took on average 25.04 ms for the 3D case and 23.42 ms for the 2D case. For the US patient data, the proposed methods were integrated into the algorithm of Crosby. Average running times on 5 datasets were: 90s, 44s, 34s and 33s for the reference, SSE, CUDA texture only and CUDA texture and shared memory methods respectively. For the reference method from the 90s, 62.50s was block matching, while 27.5s were allocated to sub-voxel displacement estimation and regularization purposes.

IV. Discussion and Conclusions

The choice of 4x4 subdivision of the VOI plane was a consequence of the fact that typical SAD kernels have a size smaller than 16 voxels in all dimensions, thus a 16x16 division was not possible. However since the kernel VOI is reused several times, a rearranging of the 4x4 values into a contiguous 16x1 block is feasible and that should further reduce computation costs since only one read operation per 4x4 kernel square, instead of four, would be required.

The advantage of CUDA is its robustness, further speed-up by taking into account memory coalescing is theoretically

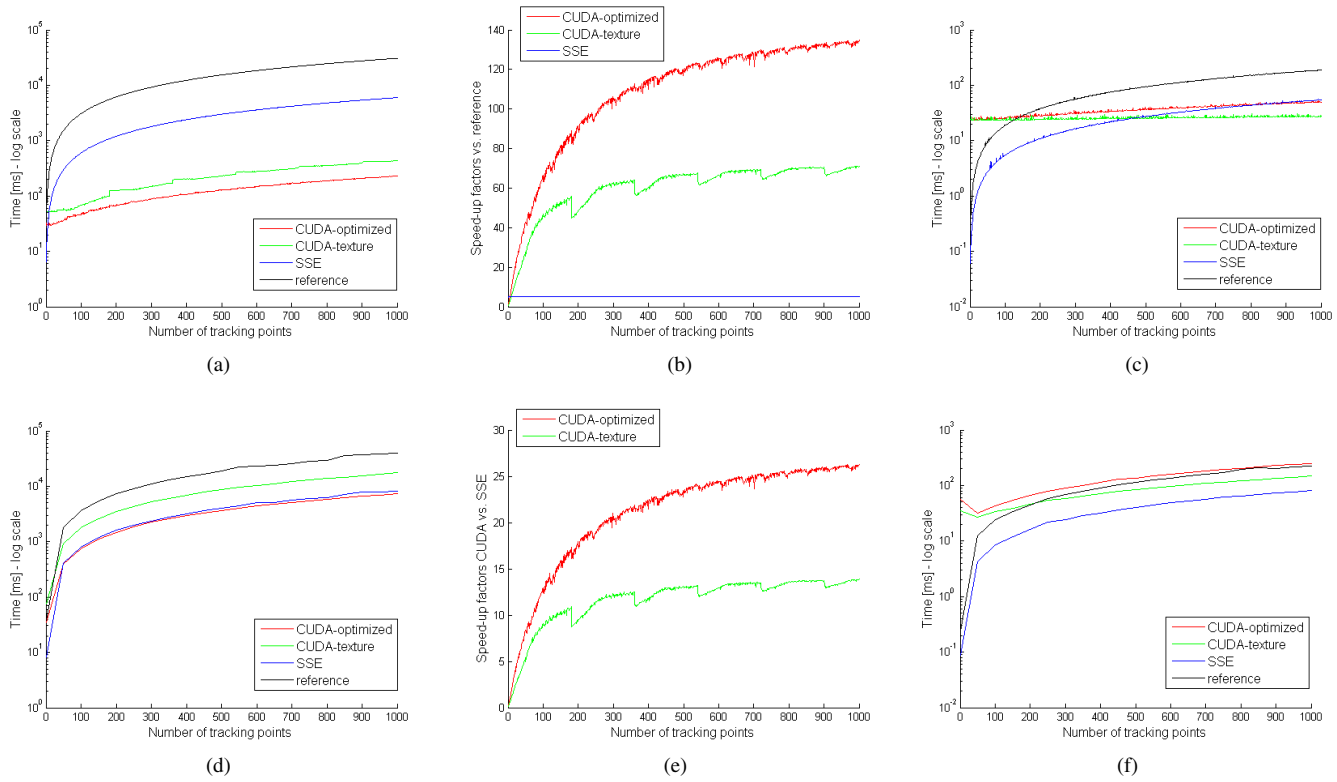

Figure 4: Results for the block matching process. Run time for the 3D case: (a) desktop, (d) laptop. Speed-up factors: (b) all methods versus reference (e) CUDA methods versus SSE. Run time for the 2D case: (c) desktop, (f) laptop.

achievable. Other matching metrics such as SSD and normalized cross-correlation can be implemented without much computational overhead. Their implementation using SSE is much more cumbersome due to the lack of optimized instructions for multiplication operations.

The performance of CUDA in 2D block-matching is severely limited by the transfer of the data to the GPU memory. However if the data is transferred anyhow, e.g. for rendering purposes, it would be highly beneficial to use CUDA for block matching purposes. Since the kernel size is much smaller in 2D, transferring it to shared memory does not have such an important impact. In fact, using only the texture units (that are optimized for 2D textures) for accessing the data seems to be more advantageous.

Apart from the block matching process, the algorithm of Crosby includes a regularization step as well, that is why speed-up factors reported on patient data are smaller than the ones for synthetic data. At the moment the CUDA implementation uses a fixed kernel and search VOI for all tracking points. However the maximum tissue velocity varies along the LV surface, this information can be exploited to reduce the size of the search VOI in regions that have small velocity values.

With the use of the SIMD architecture, the overall processing time of block matching is significantly reduced. This can lead to 3D speckle tracking methods fast enough for a clinical setting.

REFERENCES

[1] Goffinet Cline and Vanoverschelde Jean-Louis. Speckle tracking echocardiography. *European Cardiovascular Disease*, 2007.

[2] L. Hatle and G. R. Sutherland. Regional myocardial function–a new approach. *Eur Heart J*, 21(16):1337–1357, Aug 2000.

[3] J. D'hooge, A. Heimdal, F. Jamal, T. Kukulski, B. Bijnens, F. Rademakers, L. Hatle, P. Suetens, and G. R. Sutherland. Regional strain and strain rate measurements by cardiac ultrasound: principles, implementation and limitations. *Eur J Echocardiogr*, 1(3):154–170, Sep 2000.

[4] P. van Dijk. Direct cardiac nmr imaging of heart wall and blood flow velocity. *J Comput Assist Tomogr*, 8(3):429–436, Jun 1984.

[5] J. Meunier. Tissue motion assessment from 3d echographic speckle tracking. *Phys Med Biol*, 43(5):1241–1254, May 1998.

[6] A. A. Morsy and O. T. Von Ramm. 3d ultrasound tissue motion tracking using correlation search. *Ultrason Imaging*, 20(3):151–159, Jul 1998.

[7] Fredrik Orderud, Gabriel Kiss, Stian Langeland, Espen Remme, Hans Torp, and Stein Inge Rabben. Combining edge detection with speckle-tracking for cardiac strain assessment in 3d echocardiography. In *Proceedings of the IEEE Ultrasonics symposium - IUS 2008*, 2008.

[8] Jialin Song, Chunlei Li, Chun Tong, Haoyi Yang, Xia Yang, Jie Zhang, and Youbin Deng. Evaluation of left ventricular rotation and twist using speckle tracking imaging in patients with atrial septal defect. *J Huazhong Univ Sci Technolog Med Sci*, 28(2):190–193, Apr 2008.

[9] Jonas Crosby, Brage H Amundsen, Torbjørn Hergum, Espen W Remme, Stian Langeland, and Hans Torp. 3-d speckle tracking for assessment of regional left ventricular function. *Ultrasound Med Biol*, 35(3):458–471, Mar 2009.

[10] An Elen, Hon Fai Choi, Dirk Loeckx, Hang Gao, Piet Claus, Paul Suetens, Frederik Maes, and Jan D'hooge. Three-dimensional cardiac strain estimation using spatio-temporal elastic registration of ultrasound images: a feasibility study. *IEEE Trans Med Imaging*, 27(11):1580–1591, Nov 2008.

[11] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948, 1972.

[12] Mostafa Hagog. Looking for 4x speedups? sse to the rescue! [http://noel.feld.cvut.cz/vyu/scse/sse.pdf]. Technical report, Microprocessor Technology Labs, Intel Corporation, 2007.