Soft Comput (2013) 17:1095–1106
DOI 10.1007/s00500-012-0956-8

METHODOLOGIES AND APPLICATION

Code reordering using local random extraction and insertion (LREI) operator for GPGPU-based track-before-detect systems

Przemysław Mazurek

Published online: 29 November 2012

Abstract Track-before-detect (TBD) algorithms are used for tracking systems, where the object's signal is below the noise floor (low-SNR objects). A lot of computations and memory transfers for real-time signal processing are necessary. GPGPU in parallel processing devices for TBD algorithms is well suited. Finding optimal or suboptimal code, due to lack of documentation for low-level programming of GPGPUs is not possible. High-level code optimization is necessary and the evolutionary approach, based on the single parent and single child is considered, that is local search approach. Brute force search technique is not feasible, because there are $N!$ code variants, where N is the number of motion vectors components. The proposed evolutionary operator—LREI (local random extraction and insertion) allows source code reordering for the reduction of computation time due to better organization of memory transfer and the texture cache content. The starting point, based on the sorting and the minimal execution time metric is proposed. The unbiased random and biased sorting techniques are compared using experimental approach. Tests shows significant improvements of the computation speed, about 8 % over the conventional code for CUDA code. The time period of optimization for the sample code is about 1 h (1,000 iterations) for the considered recursive spatio-temporal TBD algorithm.

Communicated by A.-A. Tantar.

P. Mazurek (✉)
Department of Signal Processing and Multimedia Engineering,
West-Pomeranian University of Technology,
26. Kwietnia 10 Str., 71-126 Szczecin, Poland
e-mail: przemyslaw.mazurek@zut.edu.pl

Keywords Evolutionary programming · Code optimization · GPGPU · Track-before-detect algorithm

1 Introduction

Tracking systems are very important for surveillance applications (Blackman and Popoli 1999). Tracking of the missiles, ships, airplanes, near-Earth asteroids (NEO), and ground surface objects are typical applications. There are numerous tracking filters that are used successfully, such as Benedict–Bordner (Brookner 1998), Kalman (1960), and EKF (Blackman and Popoli 1999). More advanced tracking filters, such as Bayesian filter (Stone et al. 1999) and derivatives are also applied, if non-linear effects and non-Gaussian noises occur (Stone et al. 1999).

Most tracking systems use the detection and tracking scheme (Fig. 1) (Blackman and Popoli 1999). The object is tracked when is properly detected. The signal level of the object should be over the background noise floor. The threshold signal processing algorithms are applied for the object detection and further the estimation of the position.

The distance between signal level, related to object and background, is variable due to variable characteristic of the signal of the object, measurement conditions, and properties of the acquisition system in the real applications. Application of the tracking filter that is used as a predictor allows the improvement of the detection ratio (Blackman 1986; Blackman and Popoli 1999).

The predicted values reduce the detection area using the gate technique (Bar-Shalom 1992; Blackman 1986; Blackman and Popoli 1999; Brookner 1998), which is computationally important. Moreover, the restoration of the object's state (position, velocity) is possible when a

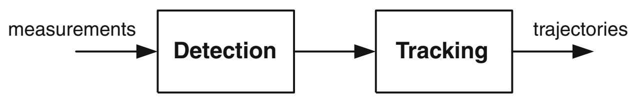


Fig. 1 Detection and tracking scheme

signal is weak. The implementation of tracking systems for high SNR (signal-to-noise ratio) cases is rather simple. The multiple target tracking systems are more sophisticated, because advanced assignment algorithms are necessary for the track maintenance (assignment of observations to the proper trajectories). Tracking filters and assignment algorithms improve tracking for a lower SNR cases (Bar-Shalom 1992; Blackman and Popoli 1999).

1.1 Outline of the paper

Very interesting, from the application point-of-view, is the case where the object signal is low, even below the noise floor (SNR < 1). The signal hidden in a noise is not detectable using a fixed or adaptive threshold algorithms. Fortunately, such signals are detected and tracked using the opposite scheme: Track-before-detect (TBD) that is considered briefly in Sect. 2. The spatio-temporal TBD code implementation techniques and computation cost are emphasized in Sect. 2. Markov matrix (sparse matrix) computations are implemented by the set of MAC operations, because regular matrix multiplication is inefficient. The advantages and limitations of the parallel processing of TBD algorithms using GPGPU (General-Purpose Graphics Processing Unit) are considered in Sect. 3. GPGPU code optimization is necessary for the processing time reduction of TBD algorithm. The proposed optimization technique for reordering of the source code using the introduced 'local random extraction and insertion' (LREI) operator is considered in Sect. 4. Reordering is a well-known technique for assembly level optimization and is based on the metric for the execution of assembly code and processing units constraints. Particular GPGPU implementation uses a high-level language (C-like) for not well documented GPGPU architecture.

1.2 Contribution-LREI operator

Introduced in this paper, the LREI operator allows optimization for such case and the execution time is reduced about 8 %, typically. The experimental results are presented in Sect. 5. Optimization technique needs appropriate starting point. The selection of starting point is evaluated in Sect. 5 for random and sorting-based techniques. Tests (benchmarks) for unbiased and biased starting points are based on the Monte Carlo approach for the reliable comparison of results and unbiased conclusions.

Theoretical evaluation of the optimization techniques for contemporary GPGPU architectures is not possible without documentation. The proposed approach shows how to optimize automatically GPGPU code without this documentation (local metrics), what is important for software developers.

1.3 Related works

Proposed technique for CUDA is based on the previous experiences, related to the code and algorithm optimization techniques (separated and combined). Optimization of the ST-TBD code is possible using specific changes in algorithm due to processing architecture. The best way for improvement of ST-TBD is the downsampled approach that increases computation speed up to 6 times (Mazurek 2010a, c). Automatic code profiling using search for the optimal processing block size changes computation speed up to few times (Mazurek 2010b). Selection of the proper memory for the state space and measurement data (texture memory instead global memory) gives few percent improvements (Mazurek 2009b).

This work is related to the adaptive compilers (Cooper et al. 2002), also. The adaptive compilers do not prefer code optimization using fixed heuristics (Joseph et al. 2008; Kisuki et al. 2000), like typical compilers. The hill climbing (random local search technique) and evolutionary search (e.g. GA, Cooper et al. 1999) techniques are used, typically. It is indicated in Almagor et al. (2004) and Kulkarni et al. (2007) that the hill climbing solution is very close to the global optimum within a small number of steps. The proposed LREI operator is a kind of the hill climbing dedicated to the code line reordering.

2 Track-before-detect systems

2.1 Introduction

Low-SNR tracking cases are very important for modern tracking systems (Blackman and Popoli 1999; Stone et al. 1999). The signal of the object is reduced due to the larger distance from the acquisition system. Moreover, the 'stealth' and countermeasure techniques are used for intentional reduction of the object's signal. A weak signal occurs for civil applications, because poor atmospheric conditions reduce effective range of sensors, which is important in transportation surveillance systems.

Detection and estimation of signals hidden in noise are important due to physical limitations of the sensors. The extending of the detection and tracking range, using algorithmic way, is very important for contemporary applications.

The track-before-detect (TBD) scheme (Fig. 2) (Blackman and Popoli 1999; Boers et al. 2008; Doucet et al.

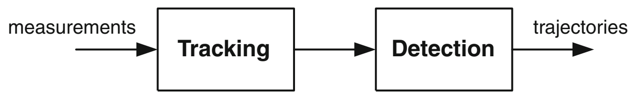


Fig. 2 Track-before-detect scheme

2001; Ristic et al. 2004; Stone et al. 1999) is used for the tracking a low SNR objects. The opposite processing order in TBD algorithms is used. This scheme assumes that tracked object is in every possible state (position, velocity, etc.) and is tracked using all expected trajectories.

The incoming signal values are accumulated over trajectories in simple TBD algorithms, for example. Such technique reduces a noise, improves SNR, and gives the ability of detection for the proper trajectory. Other tested trajectories not related to the object are also filtered and for the Gaussian noise. The mean value for the object's trajectory is much larger in comparison to the practically zero valued other trajectories.

2.2 Spatio-temporal TBD algorithm

There are many TBD algorithms, and the spatio-temporal TBD algorithm is very interesting for practical applications. This algorithm is a kind of the multidimensional IIR filter (Mazurek 2009b, 2010a, c).

New measurements are applied in the information update formula (3). The information update formula is a kind of the exponential smoothing filter. The smoothing coefficient affects the mixing between previous state space predictions and new incoming data. Larger values of smoothing coefficient, near to 1.0 for low SNR scenarios are used.

The predicted values are computed using the motion update formula (2) and the trajectories are defined by the Markov transition matrix. The prediction is based on the previous results of the information update formula. The information update formula sharps a state space values, and the motion update formula blurs state space values. Both formulas should be balanced for the reasonable tracking system. The state space is 4D for 2D input images and 2D motion vectors.

The following pseudocode shows this algorithm:

```
Start
$P(k = 0, s) = 0$    (1)
For $k \geq 1$ and $s \in S$
    $P^{-}(k,s) = \int_{S} q_k(s|s_{k-1}) P(k-1, s_{k-1}) \mathrm{d}s_{k-1}$    (2)
    $P(k,s) = \alpha P^{-}(k,s) + (1-\alpha) X(k)$    (3)
EndFor
End
```

where Eq. (1) is the initialization, Eq. (2) the motion update, Eq. (3) the information update, S the state space, e.g. 2D position and motion vectors, s the state (spatial and velocity components), k the step number or time moment (integer values), α the smoothing coefficient $\alpha \in (0, 1)$, $X(k)$ the measurements (input image), $P(k, s)$ the estimated value of objects, $P^{-}(k, s)$ the predicted value of objects and $q_k(s|s_{k-1})$ is the state transitions (Markov matrix).

Spatio-temporal TBD algorithm is computationally demanding (Mazurek 2010a, c). All possible trajectories are processed even if no object is in the range (Stone et al. 1999). The important property of this algorithm is multiple targets tracking possibility without additional costs. Efficient implementations are necessary, especially for real-time systems.

2.3 Computational requirements

Considering quite simple TBD system: input image 1,000 $\times$ 1,000 resolution, 13 motion vectors, there are 13 M state space cells. The simplest Markov matrix does not use transitions between state space cells, and there is no image blur in the motion update formula. Assuming 100 frames per seconds image rate, there are 1.3 G/s accumulations and 2.6 G/s multiplications.

The cooperation between trajectories is necessary for tracking systems, hence computed valued for state space cell should be used by the surrounding state space cells also. This is the explanation of the blurring effect in the motion update formula and it is necessary for reasonable size of the state space for maneuvering objects. The number of trajectories is fixed and there are possible trajectories of the object that are not well fitted to the available set, so co-operation is necessary.

The Markov matrix is not implemented using conventional matrix multiplication. This matrix is very large and sparse, also. Implementations are based on the embedding of the Markov transitions into code directly. It is a fast technique that reduces occupancy of the data bus and memory. The Markov matrix has similar values often, so additional code optimization is possible, especially the reduction of multiplications by the constant values. Design and optimization techniques for the Markov matrix are not considered in this paper.

2.4 Processing devices for TBD algorithms

The possible processing devices for TBD algorithms are: custom VLSI chips, FPGAs, GPGPUs, DSPs and SIMD-based CPUs. Custom chips are very interesting, but available for specific applications, especially military. FPGAs chips are well fitted for efficient implementations but they need careful synthesis and verification. GPGPUs are most important for TBD systems, because such chips are available at low cost (modern graphic cards). The cost reduction is important for civil applications and it is significant motivation for the author. DSPs- and SIMD-based CPUs are similar to GPGPUs but the level of parallelism is lower. GPGPUs support hundreds of processing cores (Kirk and Hwu 2010; NVIDIA 2011a, b) what is important for processing the motion update formula with many possible transitions between trajectories.

2.5 Organization of memory for ST-TBD algorithm

The computation of the information update formula is simple because the 2D image is added to the state space using the exponential smoothing and this operation is not considered in this paper in detail. Assuming $N \times N$ input image size and M as a number of motion vectors there are $N \times N \times M$ exponential smoothing operations.

Conventional image processing filtering techniques (a set of first order IIR filter) could be applied for the specific state space formulation. It is assumed that set of 2D motion vectors corresponding to the input image coordinates, and additional dimension for the motion vector number are used. Such formulation of the state space is important due to efficient implementations for all mentioned processing devices. Optimized 2D software routines or hardware structures are available for all of them.

The motion update formula needs much more operations. The size of the state space depends on the expected tracking resolution. The resolution of the state space that corresponds to the input image is a typical case, but higher resolutions are possible for super-resolutional tracking. The case with the $N \times N$ of position cells and M of motion vectors is typical and allows the visualization of the tracking process (Mazurek 2009a). The memory organization for TBD system and input images is shown in Fig. 3.

Organizing of the state spaces on the 2D plane for visual inspection of the results is possible. It is important for the human operator of the tracking system (Mazurek 2009a). Assuming fixed number of weight coefficients between every pair of motion planes as K, there are $N \times N \times M \times K$ of 'Multiply and ACcumulate' (MAC) operations.

The number of MAC operations is increased for the fractional motion vectors. The interpolation techniques needs more MAC operations. The bilinear interpolation is applied typically, because it is supported in hardware of GPGPU.

In experimental tests the following arbitrary selected values are used: $N = 256$ image size 256×256 pixels, $M = 9$ (the number of motion vectors) and $\mathrm{M} \times K = 40$ (in this experiment there is a fixed number of motion vector, but not equally assigned to every motion vector). There are about 2.6 M of fractional motion vectors.

A few seconds are necessary for preparation of every subtest and 1,000 subtests are executed. The 1,000 limit of subtest is selected by the observation of the convergence. Every subtest is executed 100 times on GPGPU. Overall optimization for fixed motion vectors needs 100 M of TBD code runs and takes more than 1 h. Larger images, number of motion vectors occur in the real TBD systems and optimization time is longer. The assumed parameters of the TBD system are used for the code optimization researches, without very long test (e.g. few months).

3 TBD algorithm synthesis for CUDA-supported GPGPU

3.1 GPGPU programming limitations

The GPGPU devices are well suited for TBD algorithms due to low-level parallelism, necessary for the code implementation (Mazurek 2010a, c). There are few programming options that are available for the code implementation. GPGPUs may process shader languages, but shader languages are not convenient for sophisticated code writing (Kirk and Hwu 2010).

Nowadays, the NVIDIA (2011a, b) and OpenCL are used for programming of GPGPUs using C-like code. Both of them are high-level languages what is advantage for the software developer, but the efficient implementation (reduction of computation time) needs a low-level programming. The CUDA code is translated to the intermediate code—PTX (2011). The PTX code is similar to the assembly code, but it is not desired for a low-level assembly language. The PTX code is translated to low-level operations but the overall process is only known for NVidia (the manufacturer of CUDA-supported chips). Similar situation occurs for another GPGPU manufactures, unfortunately. The possibilities of particular GPGPU chips are known with only high abstraction level point-of-view (Farber 2011; NVIDIA 2011a, b; Sanders and Kandrot 2010). Detailed information about architecture, instruction set, memory management unit, and cache algorithms are not available, unfortunately.

The software developers use a set of programming rules that are defined in GPGPU and CUDA/OpenCL/PTX documentations (NVIDIA 2011a, b) and they make experiments related to the code reorganization. Some experiments fail, because the obtained code is executed too slowly. Sometimes the success is obtained and the code works faster in comparison to the previous step. Such iterative code optimization, driven by the software developer is not efficient, as a code optimization based on detailed chip documentation. Finding of the optimal order of instructions, using conventional code synthesis techniques without documentation of architecture, is not possible.

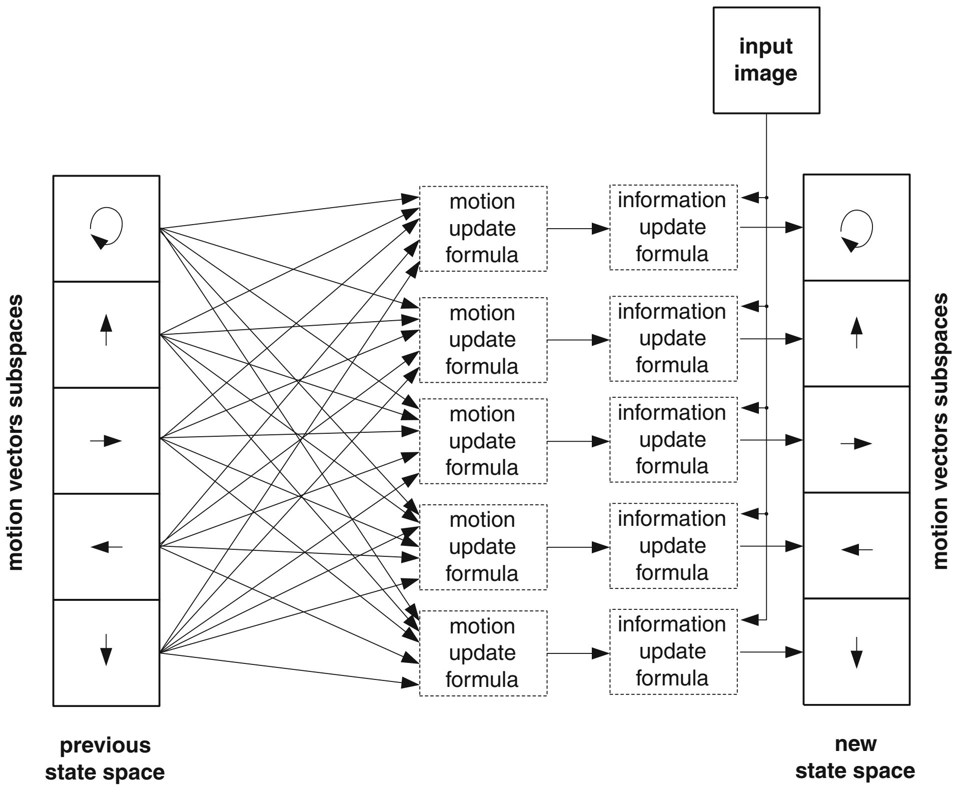


Fig. 3 Example of memory organization for TBD system

3.2 Automatic code optimization

Automatic optimization is based on the experimental approach (run and measure) and cannot be considered precisely from theoretical point-of-view.

The theoretical approach is applicable for well-defined systems with known metrics. The typical GPGPU device is "darkgray box", because only a small part of documentation is available. The theoretical approach needs complete metrics: assembly code execution time including pipeline processing, cache model, memory interfaces model, etc. Available documentation allows limited optimization, because GPGPUs are not a "black boxes" and some programming aspects are delivered (NVIDIA 2011a, b).

The lack of documentation is the typical problem for many contemporary advanced integrated circuits. Particular GPGPUs allow programming using a high-level language that is converted to the intermediate code (PTX-code). This code is converted to the real assembly code of GPGPU by the GPGPU card driver. Such approach gives compatibility of the different generation cards at PTX code level, which is very important for typical users. The changes of ISA (Instruction Set Architecture) by the chip manufacturer give abilities of the improvements of new chips, without preservation of the backward compatibility. Such strategy enables even a radical changes of ISA without recompilation of the GPGPU code of target applications (NVIDIA 2011c). Optimization of hardware processing units and ISA is very important for the reduction of the processing time and power consumption. The lack of the optimal code design is obtained, unfortunately. The assembly code is ISA-dependent code strictly, but metrics are not available, hence theoretical approach is not possible. The experimental approach is one way only, which is possible to use for software developers. Moreover, the execution for particular code cannot be obtained using single run due to reliability of the measurement timers.

Maximization of GPGPU performance is necessary for the application of computation intensive algorithms such as TBD. Automatic code synthesis or automatic code optimization is necessary with automatic performance tests. The performance tests are used in iterative search of the optimal instruction order. Other techniques such as switching between alternative memories, data organization, loop rolling/unrolling are also subjects of the automatic optimization process.

The discreet optimization techniques are the best way for solving the code synthesis problem without GPGPU architecture knowledge. The scale of the problem is well depicted for mentioned example.

The number of basic motion vectors is equal to the number of code lines of main processing code, so the calculation of the MAC and data transfers for the operation is defined as a $M \times K$ (e.g. 40) for single CUDA kernel. The number of possibilities of lines order is $(M \times K)!$, e.g. 40! for example TBD system. Brute force search techniques for all possibilities are not realistic. The more advanced search is necessary.

3.3 GPGPU constraints

CUDA-based GPGPU supports SIMT (Single-Instruction Multiple-Thread processing model) (Farber 2011; NVIDIA 2011a, b; Sanders and Kandrot 2010). Such programming model is an extended SIMD (Single-Instruction Multiple-Data) processing model. SIMT supports processing code with branches, what is not available in SIMD model. TBD algorithm is well fitted to both models, because branches are not available. The processing speed-up depends on the algorithm and implementation. Even a hundreds time faster processing is possible for specific algorithm using GPGPUs in comparison to the modern single core CPUs (x86 architecture).

Processed data are stored in the global memory (outside of GPGPU chip), and transferred between GPGPU and memory chip (a new and temporal data; the final result). The main bottlenecks of GPGPU are: the bandwidth and latency limits, related to the memory transfers between the GPGPU and memory chip. The reduction of both bottlenecks using internal shared memory, texture cache and constants memory is necessary (NVIDIA 2011a, b).

The data transfer intense algorithms and the large memory sets are not well fitted into GPGPU architecture. TBD algorithms belong to both groups, unfortunately. TBD algorithms are slow without special optimization techniques, but the efficient algorithm level optimization techniques are available (Mazurek 2010a, b, c, 2011). Instruction code and memory-related optimization techniques are also proposed, e.g. in (Mazurek 2010b, c).

One of the most interesting techniques is the application of texture unit that supports memory transfers from the global memory, a small cache and the acceleration of the bilinear interpolation. The bilinear interpolation allows computation values using a non-integer memory addresses, what is useful for TBD algorithm using proposed organization of the state space memory. The cache memory improves read operations from memory (Farber 2011; NVIDIA 2011a, b; Sanders and Kandrot 2010) and is an alternative to the custom memory management using the shared memory using an additional code. The best solution is based on the utilization of both GPGPU capabilities: the texture cache and the shared memory. Shared memory, used as a temporal storage of state space values, allows the reduction of non-coalescence write operations (Farber 2011; NVIDIA 2011a, b; Sanders and Kandrot 2010). Local operations on the state space area correspond to the local memory operations. Shared memory is used for the temporal result storage and allows simultaneous write operation for neighborhood location (Farber 2011; NVIDIA 2011a, b; Sanders and Kandrot 2010).

4 Optimization technique for TBD code

4.1 Introduction

The evolutionary (Back et al. 2000a, b; Michalewicz 1998; Spears 2000) technique, using the proposed operator, reduces the computation time of TBD algorithm. The proposed LREI operator could be considered as local search operation or as a kind of the evolutionary operator from evolutionary perspective—it is equivalent to transposition in genetics. Code lines reordering is a well-known assembly code optimization technique. Different order of instruction execution, without changes of results, is used by the optimizing compilers or by the processors directly. The first variant is based on the defined metrics. Integer linear programing (optimization) techniques are applied typically. Some processor architectures, such as Pentium 4, can change order of execution, also. The texture unit is not well documented, but the optimization technique operator is based on the general knowledge about cache and memory bus.

High-level (e.g. C-language) code optimization is possible depending on the compiler. Some compilers use optimization techniques and the order of execution cannot be selected by the software developer. Some compilers allow such operation, fortunately.

Reordering of the CUDA instructions is efficient technique that improves texture cache utilization and reduces the global memory transfer. Such optimization techniques are not well described in the literature. Most GPGPU optimization techniques are related to the parallel implementation of algorithm.

4.2 Structure of the motion update code

The ST-TBD code has quite simple structure. First part is used for the calculation of the motion update formula using MAC operations. The memory transfers of the state space from the global memory via the texture cache are used. The shared memory is used for the temporal result storage. After all operation in the first part, new input values update temporal results. It is second part (coefficient does not influence the computation time). Third part is related to the state space to the global memory transfer. Two last parts are not subject of optimization.

The following code line is responsible for basic motion vector (first part):

$$V = V + W \cdot tex2D(texImage, x + SmallOffsetX, \\ y + SmallOffsetY + MotionVectorOffset) \quad (4)$$

and the example code line (part of the Markov transitions) looks like:

$$\mathrm{X4} = \mathrm{X4} + 0.804\mathrm{f} * \mathrm{tex2D(texImage,x} + (-1.90\mathrm{f}), \\ \mathrm{y} + (-0.55\mathrm{f}) + 4 * \mathrm{OFFSET}) \quad (5)$$

There are 40 lines like this for different motion vectors in example code. Transfers from memory via texture unit are based on CUDA 'tex2D' function (NVIDIA 2011a, b). The *texImage* is the reference of the texture memory block. The spatial position is defined using the integer image coordinates x and y. The bilinear interpolation is used for non-integer offsets by the applications of *SmallOffsetX* and *SmallOffsetY* fractional part. Four memory values are necessary for the calculation of the bilinear interpolation result, and two or four memory transfers are necessary. Efficient implementation should process data using a surrounding address values '$x + SmallOffsetX$' and '$y + SmallOffsetY$'. The *MotionVector Off set* is used for addressing the particular motion subspace. The V is the temporal variable and the W is the constant weight coefficient. All values are floating point numbers (32-bit wide). The number of motion vectors is defined by the number of temporal variables.

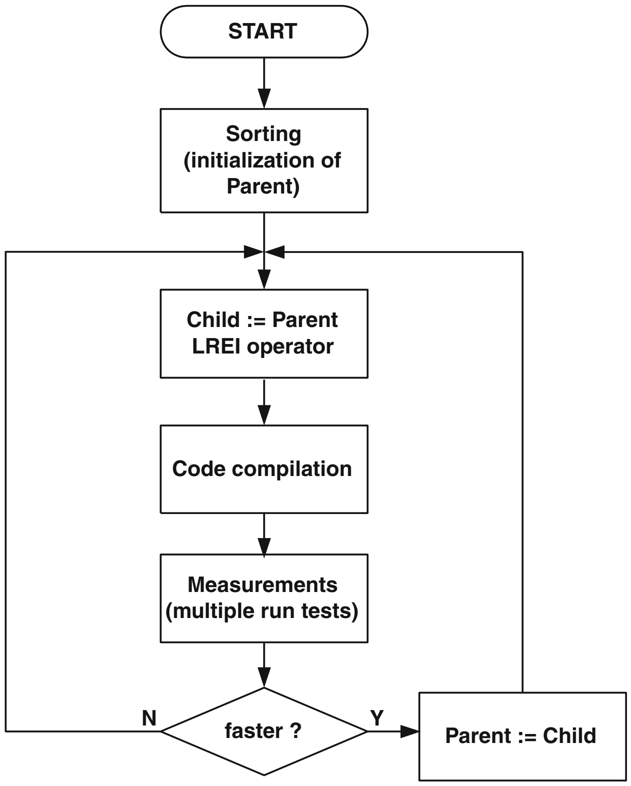

START
Sorting (initialization of Parent)
Child := Parent LREI operator
Code compilation
Measurements (multiple run tests)
faster ?
N
Y
Parent := Child

Fig. 4 Main processing steps

The subject of optimization is the list of the MAC operations with texture transfer functions. There are no additional constraints about the order of instructions related to the algorithm, what is very promising. Main processing steps are depicted in Fig. 4.

4.3 Solutions for the starting point problem

The starting point for any optimization technique is very important. A well estimated starting point may reduce optimization steps and improves convergence to the sub-optimal or optimal solution.

The set of the code lines is sorted using the following order: *SmallOffsetX*, *SmallOffsetY*, and *MotionVectorOffset*. This order is related to the address distance between.

Largest distance is related to the *MotionVector-Offset*, medium distance to the *SmallOffsetY*, and local distance to the *SmallOffsetX*.

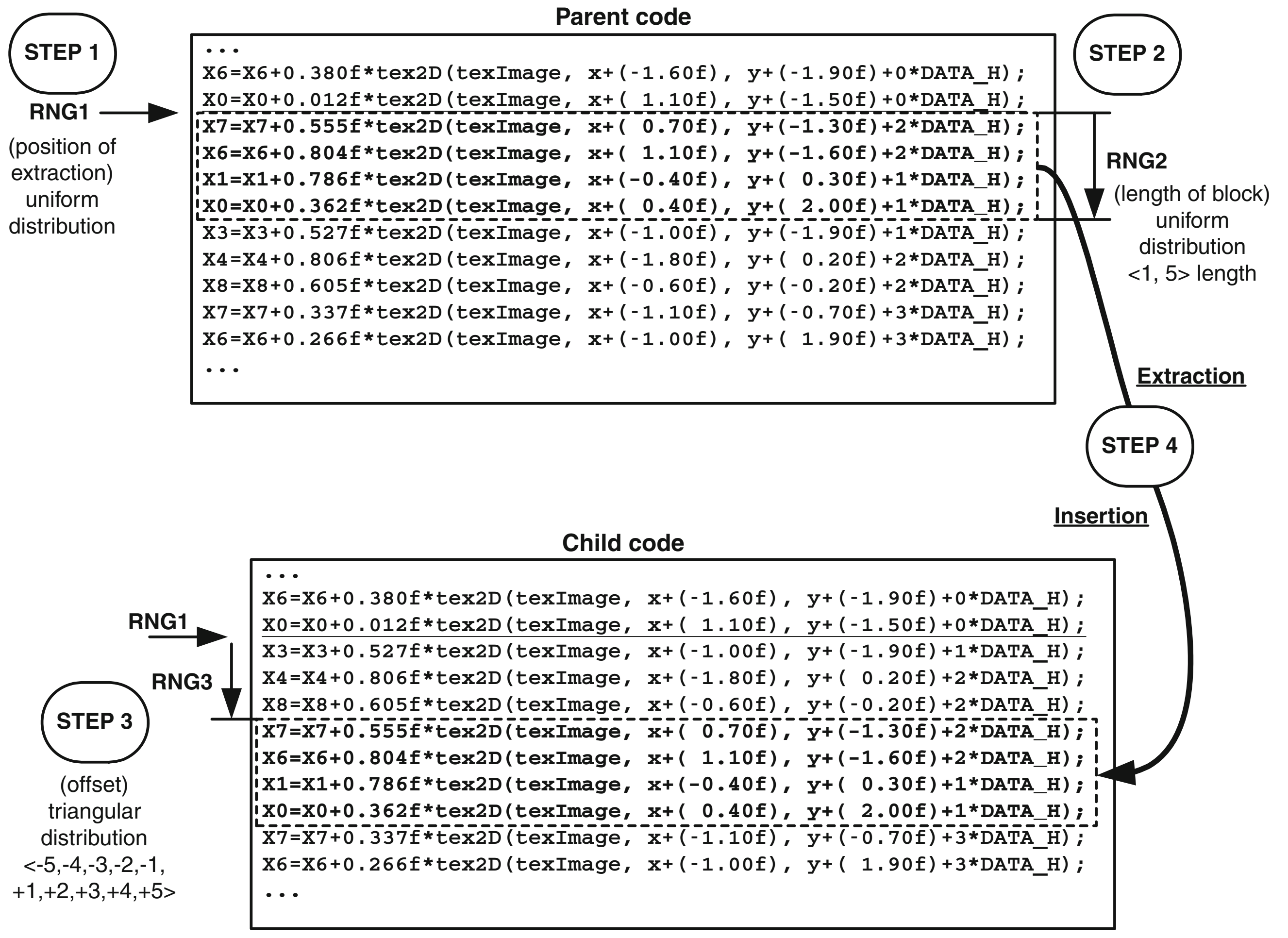
Experimental comparison of the random and sorting-based techniques for the starting point is presented in Sect. 5.

4.4 Solution for the metric of code problem

The real computation time is assumed as the metric of the code and the optimization error criteria. Minimization aim is assumed—the reduction of the computation time. The optimization process finishes after selected number of iterations (subtests).

Measurements of the computation time are not simple and trivial. There are timer-based functions that are used for estimation of code execution. It should be emphasized that it is only estimation, due to time measurement errors, not a exact value of execution time. Multiple code runs give different time values. One of the most important factors that influence the results is the preemptive behavior of operating systems. Microsoft Windows and Linux support preemptive process switching. CUDA execution time measurement could be extended by the another processes assigned to CPU. Single run is not sufficient, because time period values are sometimes a few times higher. Multiple runs and calculation of the mean value are much more reliable. The processing time has Gaussian probability curve typically, but the right side long tail may occur. Preferred metric should be based on the median value from multiple runs not on the mean value. Such robust estimator eliminates influence of long tails that is typical for the mean estimator.

Median and mean values are influenced by the external programs and are not reliable as a metric for the code optimization. The minimal execution time is much better,


Parent code
STEP 1
RNG1
(position of extraction) uniform distribution
STEP 2
RNG2
(length of block) uniform distribution <1, 5> length
Extraction
STEP 4
Insertion
Child code
RNG1
RNG3
STEP 3
(offset) triangular distribution <-5,-4,-3,-2,-1, +1,+2,+3,+4,+5>

```
...
X6=X6+0.380f*tex2D(texImage, x+(-1.60f), y+(-1.90f)+0*DATA_H);
X0=X0+0.012f*tex2D(texImage, x+( 1.10f), y+(-1.50f)+0*DATA_H);
X7=X7+0.555f*tex2D(texImage, x+( 0.70f), y+(-1.30f)+2*DATA_H);
X6=X6+0.804f*tex2D(texImage, x+( 1.10f), y+(-1.60f)+2*DATA_H);
X1=X1+0.786f*tex2D(texImage, x+(-0.40f), y+( 0.30f)+1*DATA_H);
X0=X0+0.362f*tex2D(texImage, x+( 0.40f), y+( 2.00f)+1*DATA_H);
X3=X3+0.527f*tex2D(texImage, x+(-1.00f), y+(-1.90f)+1*DATA_H);
X4=X4+0.806f*tex2D(texImage, x+(-1.80f), y+( 0.20f)+2*DATA_H);
X8=X8+0.605f*tex2D(texImage, x+(-0.60f), y+(-0.20f)+2*DATA_H);
X7=X7+0.337f*tex2D(texImage, x+(-1.10f), y+(-0.70f)+3*DATA_H);
X6=X6+0.266f*tex2D(texImage, x+(-1.00f), y+( 1.90f)+3*DATA_H);
...
```

```
...
X6=X6+0.380f*tex2D(texImage, x+(-1.60f), y+(-1.90f)+0*DATA_H);
X0=X0+0.012f*tex2D(texImage, x+( 1.10f), y+(-1.50f)+0*DATA_H);
X3=X3+0.527f*tex2D(texImage, x+(-1.00f), y+(-1.90f)+1*DATA_H);
X4=X4+0.806f*tex2D(texImage, x+(-1.80f), y+( 0.20f)+2*DATA_H);
X8=X8+0.605f*tex2D(texImage, x+(-0.60f), y+(-0.20f)+2*DATA_H);
X7=X7+0.555f*tex2D(texImage, x+( 0.70f), y+(-1.30f)+2*DATA_H);
X6=X6+0.804f*tex2D(texImage, x+( 1.10f), y+(-1.60f)+2*DATA_H);
X1=X1+0.786f*tex2D(texImage, x+(-0.40f), y+( 0.30f)+1*DATA_H);
X0=X0+0.362f*tex2D(texImage, x+( 0.40f), y+( 2.00f)+1*DATA_H);
X7=X7+0.337f*tex2D(texImage, x+(-1.10f), y+(-0.70f)+3*DATA_H);
X6=X6+0.266f*tex2D(texImage, x+(-1.00f), y+( 1.90f)+3*DATA_H);
...
```

Fig. 5 Example of LREI operator processing steps

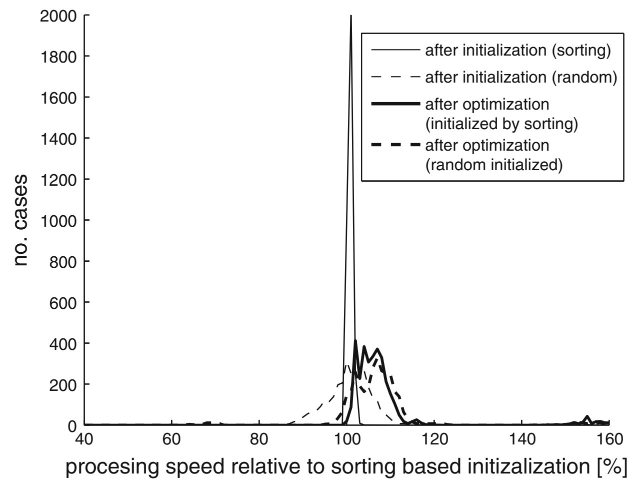


Fig. 6 Comparison of performance after 100 iteration for random and sorting-based initialization

because the differences are related to the host platform (CPU), not GPGPU. Multiple code runs are still necessary.

The influences on time measurements from the host platform are reduced, if the execution time of CUDA code is longer. Such time values are more reliable, but the overall optimization process is proportionally longer, unfortunately.

4.5 LREI (local random extraction and insertion)

Selection of the optimization technique is very important for convergence. The assumed evolutionary technique uses single parent and single child. New parent is established if the child is faster in comparison to the current parent. Local search is used until maximal number of iteration is reached.

The aim of the LREI operator is to reorder locally a part of the code. The code is moved from one point (extracted) to another one (inserted) that is depicted in Fig. 5. The position of extraction is driven by the uniform RNG1 (Random Number Generator). The length of extraction is driven by the uniform RNG2:

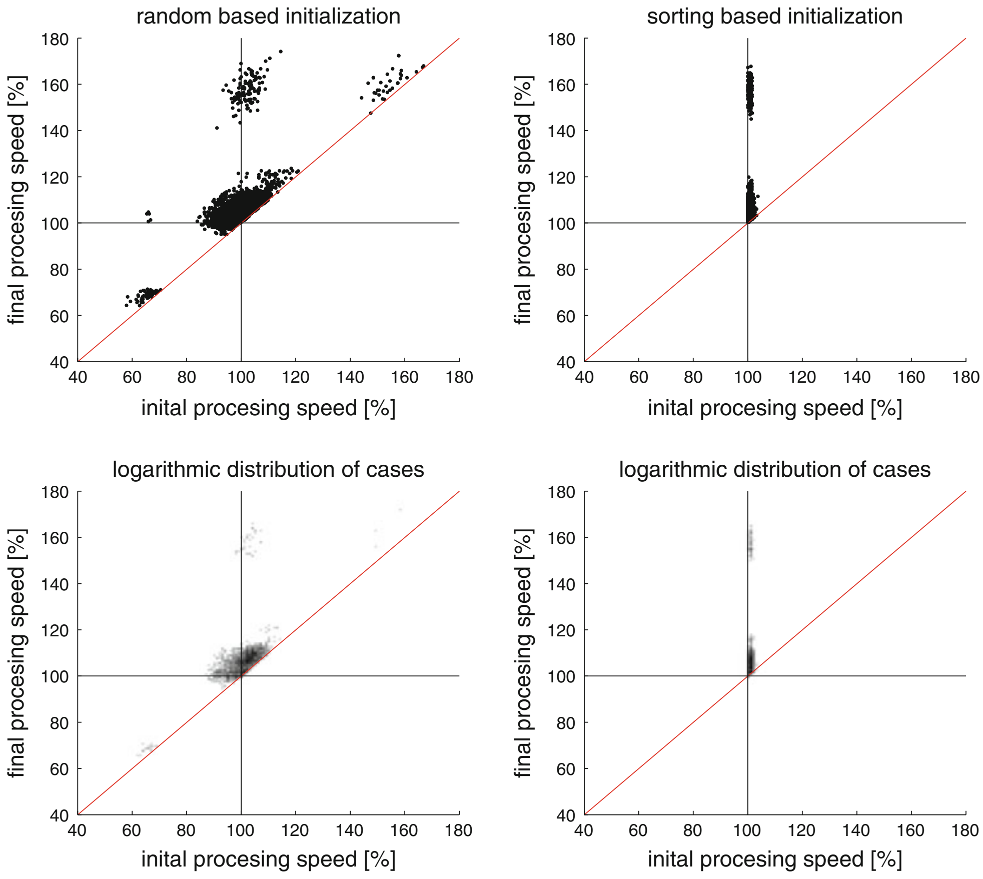


Fig. 7 Processing speed: random (*left*) and sorting-based (*right*) initialization (after initialization and after 100 iterations)

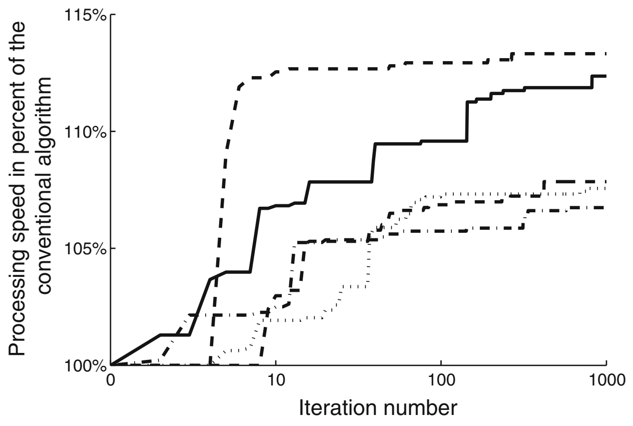


Fig. 8 Changes of computation speed depending on iteration step—every curve is an optimization example, for different starting points and code lines motion vectors

$$\mathrm{RNG2} \in \{1, 2, 3, 4, 5\}. \qquad (6)$$

The position of insertion is driven by the RNG3 with triangular distribution:

$$\mathrm{RNG3} \in \{-5, -4, -3, -2, -1, +1, +2, +3, +4, +5\} \qquad (7)$$

The 0 offset value of RNG3 does not change position of the insertion point, so this value is not used.

5 Experimental results

5.1 Comparison of the random and sorting-based initializations

The G84 GPGPU core is used in tests (GeForce 8600 GTS) and Intel Pentium 4D 2.6 GHz, 1 GB RAM, Debian Linux 3.0 amd64.

The texture cache memory that supports bilinear interpolation and global memory paging prefers local data operation. Distant address of read operations reduces the performance of the system. This is the reason of the selection of local changes using LREI operator and selection of the sorting algorithm for initialization that is probably quite near to the global optima.

The sorting order is:

$$\begin{aligned}&SORT(increasingorder):\\&\quad\{SmallOffsetY, SmallOffsetX, OFFSET\}\end{aligned} \qquad (8)$$

because *OFFSET* selects distant addresses, so it is last. The selection of the *SmallOffsetY*, *SmallOffsetX* or opposite *SmallOffsetX*, *SmallOffsetY* order is necessary for the reduction of local address distance between two following code lines. The sorting is not biased by the fixed *OFFSET*s.

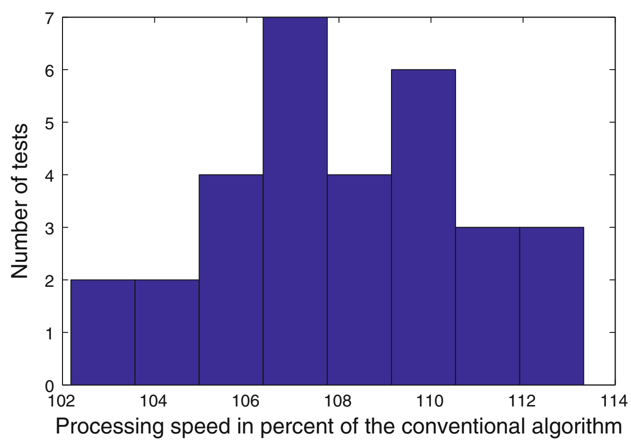


Fig. 9 Histogram of final results (after 1,000 iterations)

Typical solution which can be used by software developers, is based on the optimization of the order of code lines, using the *SmallOffsetY* and *SmallOffsetX* only. Code optimization related to *OFFSET* does not seem a promising idea. The code line order should be optimized for separate code block only. Every block should be defined by the common value of *OFFSET*.

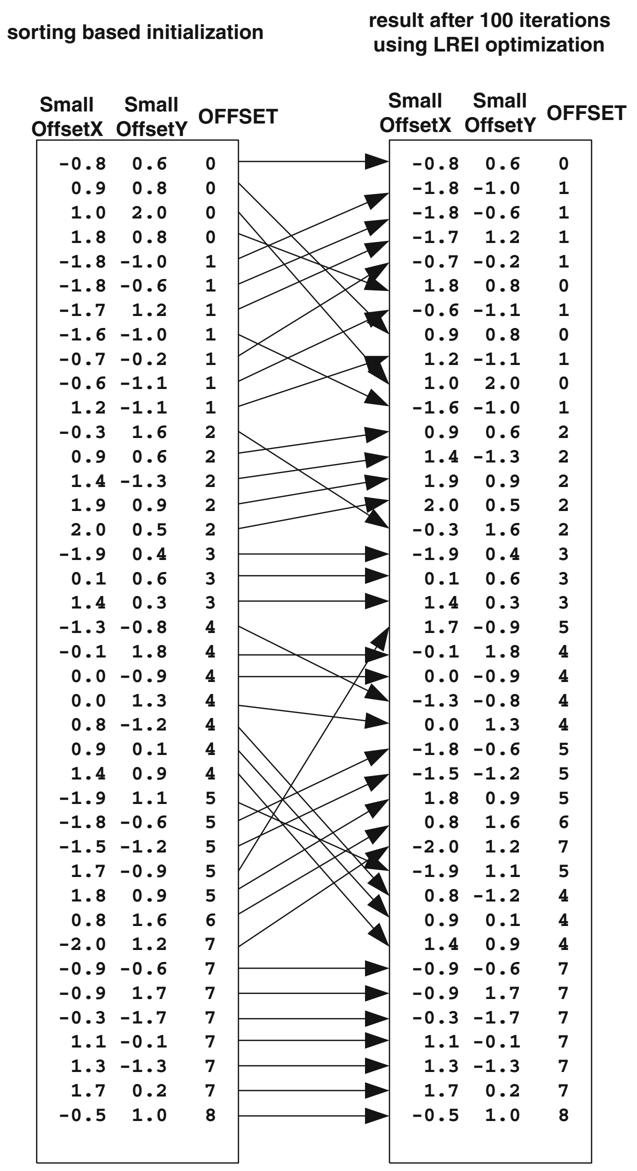
It is surprising, that such approach (based on the common knowledge) does not give best achieved results. The experiments, described later, show that better results are obtained for different orders of *OFFSET*.

The sorting algorithm forces order of operations, so position in search space is biased. This hypothesis considering starting point, selected by the sorting, should be tested against another initialization technique.

The unbiased initialization is the best reference. The random initialization for the same set of motion vectors and values is the unbiased reference. The random initialization selects starting point, near to the unknown global optima and far from this point at equal probability. Falling into local minimum is possible.

The LREI operator is based on the random generators so unbiasing of the biased starting point is obtained by many iterations steps.

The selection of the starting point strategy is tested using 100 iterations steps only. There are 20 optimization processes for the sorting strategy (all of them have identical starting configuration). There are also 20 optimization processes for the random initialization strategy. The number of iterations and number of processes is selected due to very long execution time of the Monte Carlo test. Such test, for the particular motion vector set, takes about 3.5 h. There are 170 tests for different motion vectors, so all tests take about 600 h per single computer (25 days). Few computers were used for reduction of processing time in this test.


sorting based initialization
result after 100 iterations using LREI optimization

sorting based initialization

Small OffsetX	Small OffsetY	OFFSET
-0.8	0.6	0
0.9	0.8	0
1.0	2.0	0
1.8	0.8	0
-1.8	-1.0	1
-1.8	-0.6	1
-1.7	1.2	1
-1.6	-1.0	1
-0.7	-0.2	1
-0.6	-1.1	1
1.2	-1.1	1
-0.3	1.6	2
0.9	0.6	2
1.4	-1.3	2
1.9	0.9	2
2.0	0.5	2
-1.9	0.4	3
0.1	0.6	3
1.4	0.3	3
-1.3	-0.8	4
-0.1	1.8	4
0.0	-0.9	4
0.0	1.3	4
0.8	-1.2	4
0.9	0.1	4
1.4	0.9	4
-1.9	1.1	5
-1.8	-0.6	5
-1.5	-1.2	5
1.7	-0.9	5
1.8	0.9	5
0.8	1.6	6
-2.0	1.2	7
-0.9	-0.6	7
-0.9	1.7	7
-0.3	-1.7	7
1.1	-0.1	7
1.3	-1.3	7
1.7	0.2	7
-0.5	1.0	8

result after 100 iterations using LREI optimization

Small OffsetX	Small OffsetY	OFFSET
-0.8	0.6	0
-1.8	-1.0	1
-1.8	-0.6	1
-1.7	1.2	1
-0.7	-0.2	1
1.8	0.8	0
-0.6	-1.1	1
0.9	0.8	0
1.2	-1.1	1
1.0	2.0	0
-1.6	-1.0	1
0.9	0.6	2
1.4	-1.3	2
1.9	0.9	2
2.0	0.5	2
-0.3	1.6	2
-1.9	0.4	3
0.1	0.6	3
1.4	0.3	3
1.7	-0.9	5
-0.1	1.8	4
0.0	-0.9	4
-1.3	-0.8	4
0.0	1.3	4
-1.8	-0.6	5
-1.5	-1.2	5
1.8	0.9	5
0.8	1.6	6
-2.0	1.2	7
-1.9	1.1	5
0.8	-1.2	4
0.9	0.1	4
1.4	0.9	4
-0.9	-0.6	7
-0.9	1.7	7
-0.3	-1.7	7
1.1	-0.1	7
1.3	-1.3	7
1.7	0.2	7
-0.5	1.0	8

Fig. 10 Example order of code lines values after initialization and after optimization

The computation time for GPGPU code depends on the set of vectors, so minimal execution time for the first iteration is the reference value (100 %). This value is obtained from 20 initializations based on the sorting strategy. All of them should be equal theoretically, but the uncertainty of measurement technique adds a small variations.

The mean value of performance speed (Fig. 6) is similar for both optimization techniques (about 108 %). Random initialization gives 41 % of starting values below initial performance of the sorting-based initialization (reference: 100 %).

There are about 7 % cases, where the random initialization does not give improvements, after 100 of iterations, over the initial performance of the sorting-based initialization (reference: 100 %). Such result shows that sorting is better in comparison to the random initialization.

Initial and obtained processing speed for every test case is shown in Fig. 7. Random initialization gives four main populations. Two of them are better on the start. There is one population, related to wrong starting point, that fall in the local minimum (left-bottom located), because the number of iterations is rather small (100). Largest population (central part of figures) has starting performance similar to the reference.

5.2 Experimental tests for larger number of iterations

The time-logarithmic figure is used for better depiction of the convergence during first iterations. Few example cases are shown in Fig. 8.

Improvements are smooth or rapid, there are also a lot iterations without improvements (Fig. 8). Starting code, based on the sorting optimization is referenced by the 100 % level value. The improvement is significant, about 8–9 % (mean value), and only slightly larger in comparison to the test use 100 iterations. The reduction of processing time is significant for first iteration steps. Longer optimization is still interesting, but is limited to the 1,000 iterations. The curves have exponential saturation shape, because there is a global limit, and the asymptotic behavior is observed. Different motion vector sets are used, so different asymptotic levels are achieved.

Final results, after 1,000 iterations, are stored and the histogram (Fig. 9) shows distribution of the results for 31 tests.

Example order of code line values are shown in Fig. 10. It is well shown that proposed optimization using LREI changes order of code lines including *OFFSET*.

6 Conclusions and further works

The idea of the application of the optimization techniques for reduction of code execution is not new. The optimization of code for efficient implementation of algorithm using optimization technique is very important. The conventional optimization approach is based on the detailed knowledge about processing architecture. Such knowledge is not available for high-performance processing devices such as GPGPUs, unfortunately. Considered technique is the useful tool for software developers and final users.

Sorting approach is based on the organization of memory. The cache memory related to the texture unit improves accesses to the neighborhood memory locations. Reordering of the code, using high-level programming language allows the reduction of computation time. It is not possible to find solution without optimization due to scale of problem.

The code optimization technique gives significant improvement of the computation time about 8 % for ST-TBD algorithm what is important for real-time applications. This is mean value, and obtained results depend on the motion vectors set. The number of iteration could be small (100) and the significant improvement is obtained in most cases.

The main problem is that optimization time is long (few hours). It is not a problem of CUDA code, but sources code compilation process. This is main bottleneck that should be considered carefully for the application of this or similar optimization techniques for CUDA code optimization.

The optimization of motion vectors (high-level optimization) and medium-level optimization (code reordering) together are the most promising method that will be considered in further work.

Acknowledgments This work is supported by the UE EFRR ZPORR project no. Z/2.32/ I/1.3.1/267/05 “Szczecin University of Technology—Research and Education Center of Modern Multimedia Technologies” (Poland).

References

Almagor L, Cooper KD, Grosul A, Harvey TJ, Reeves SW, Subramanian D, Torczon L, Waterman T (2004) Finding effective compilation sequences. Proceedings of the symposium on languages, compilers and tool support for embedded systems, pp 231–239

Back T, Fogel DB, Michalewicz Z (2000a) Evolutionary computation 1. Basic algorithms and operators. Institute of Physics Publishing, Philadelphia

Back T, Fogel DB, Michalewicz Z (2000b) Evolutionary computation 2. Advanced algorithms and operators. Institute of Physics Publishing, Philadelphia

Bar-Shalom Y (1992) Multitarget–multisensor tracking: applications and advances, vol II. Artech House, Boston

Blackman S (1986) Multiple-target tracking with radar applications. Artech House, Boston

Blackman S, Popoli R (1999) Design and analysis of modern tracking systems. Artech House, Boston

Boers Y, Ehlers F, Koch W, Luginbuhl T, Stone LD, Streit RL (2008) Track-Before-Detect algorithm. EURASIP J Adv Signal Process

Brookner E (1998) Tracking and Kalman filtering made easy. Wiley-Interscience, New York

Cooper KD, Schielke PJ, Subramanian D (1999) Optimizing for reduced code space using genetic algorithms. Proceedings of the symposium on languages, compilers and tool support for embedded systems

Cooper KD, Subramaniam D, Torczon L (2002) Adaptive compilers of the 21th century. J Supercomput 23:7–22

Doucet A, de Freitas N, Gordon N, Smith A (2001) Sequential Monte Carlo methods in practice. Springer, Berlin

Farber R (2011) CUDA application design and development. Morgan Kaufmann, Los Altos

Joseph PJ, Jacob MT, Srikant YN, Vaswani K (2008) Statistical and machine learning techniques in compiler design. In: Srikant YN, Shankar P (eds), The compiler design handbook, optimization and machine code generation. CRC Press, Boca Raton

Kalman RE (1960) A new approach to linear filtering and prediction problems. Trans ASME J Basic Eng 82(Series D):35–46

Kirk DB, Hwu WW (2010) Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, Los Altos

Kisuki T, Knijnenburg PMW, O'Boyle MEP, Wijshoff HAG (2000) Iterative compilation in program optimization. Workshop on compilers for parallel computing

Kulkarni PA, Whalley DB, Tyson GS, Davidson JW (2007) Evaluating heuristic optimization phase order search algorithms. Proceedings of the international symposium on code generation and optimization, pp 157–169

Mazurek P (2009a) Direct visualization methods for Track-Before-Detect algorithms. Pozn Univ Technol Acad J—Electr Eng 59:25–34

Mazurek P (2009b) Implementation of spatio-temporal Track-Before-Detect algorithm using GPU. Meas Autom Monit 55(8):657–659

Mazurek P (2010a) Optimization of bayesian Track-Before-Detect algorithms for GPGPUs implementations. Electr Rev R 86(7): 187–189

Mazurek P (2010b) Optimization of Track-Before-Detect systems for GPGPU. Meas Autom Monit 56(7):665–667

Mazurek P (2010c) Optimization of Track-Before-Detect Systems with decimation for GPGPU. Meas Autom Monit 56(12):1523–1525

Mazurek P (2011) Hierarchical Track-Before-Detect algorithm for tracking of amplitude modulated signals, advances in intelligent and soft computing, vol 102—image processing and communications challenges 3, Springer, Berlin, pp 511–518

Michalewicz A (1998) Genetic algorithms + data structures = evolution programs, Springer, Berlin

NVIDIA (2011a) CUDA C best practices guide v4.0. NVIDIA

NVIDIA (2011b) NVIDIA CUDA C programming guide v4.0. NVIDIA

NVIDIA (2011c) NVIDIA PTX: parallel thread execution. ISA Version 2.3. NVIDIA

Ristic B, Arulampalam S, Gordon N (2004) Beyond the Kalman filter: particle filters for tracking applications. Artech House, Boston

Sanders J, Kandrot E (2010) CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley, Reading

Spears WM (2000) Evolutionary algorithms. The role of mutation and recombination. Springer, Berlin

Stone LD, Barlow CA, Corwin TL (1999) Bayesian multiple target tracking. Artech House, Boston