Przemysław MAZUREK
WEST – POMERANIAN UNIVERSITY OF TECHNOLOGY IN SZCZECIN, DEPARTMENT OF SIGNAL PROCESSING AND MULTIMEDIA ENGINEERING
26. Kwietnia 10, 71-126, Szczecin

Optimization of Track-Before-Detect Systems with Decimation for GPGPU

Ph.D. eng. Przemysław MAZUREK

Assistant professor in the Department of Signal Processing and Multimedia Engineering at the Faculty of Electrical Engineering, West-Pomeranian University of Technology in Szczecin. Author of more than 80 papers related to the digital signal processing, estimation of object kinematics, biosignals acquisition and processing.

e-mail: przemyslaw.mazurek@zut.edu.pl

Abstract

Tracking systems based on Track-Before-Detect (TBD) scheme support tracking of low-SNR objects even if object signal is hidden in a noise. In this paper proposed method [1] is tested using Spatio-Temporal TBD algorithm with an additional code profiling using Nvidia CUDA computational platform. Different implementations are possible and the best solution for downsampled approach is based on the separate, register based state-space (without Shared Memory) and texture cache for input measurements.

Keywords: estimation, tracking, parallel image processing, GPGPU, Track-Before-Detect.

Optymalizacja systemów śledzenia przed detekcją z decymacją dla GPGPU

Streszczenie

Algorytmy śledzenia przed detekcją umożliwiają śledzenie obiektów w warunkach niskiej wartości SNR (Signal-to-Noise Ratio) jednak są one bardzo złożone obliczeniowo. Wykorzystując GPGPU (programowalny procesor graficzny) możliwa jest implementacja czasu rzeczywistego. Dla zaproponowanego w [1] rozwiązania optymalizacji implementacji algorytmu z decymacją sygnału wyjściowego możliwe jest kilkukrotne skrócenie czasu obliczeń. W artykule przedstawiono i porównano dalsze możliwe rozwiązania optymalizacji z wykorzystaniem platformy programowej Nvidia CUDA dla rekurencyjnego algorytmu Spatio-Temporal Track-Before-Detect. Przestrzeń stanów może być decymowana w celu lepszego wykorzystania szybkiej pamięci współdzielonej dostępnej w GPGPU, podczas gdy dane wejściowe oraz wyjściowe przechowywane są w wolnej pamięci globalnej. Wykorzystując testy numeryczne z wykorzystaniem opracowanego oprogramowania do profilowania kodu źródłowego stwierdzono, że najbardziej wydajnym rozwiązaniem spośród analizowanych jest implementacja z oddzielnymi kernelami przetwarzania dla poszczególnych wektorów ruchu, wykorzystania rejestrów do przechowywania danych przestrzeni stanów w miejsce pamięci współdzielonej oraz pamięci texture cache do buforowania danych wejściowych. W przypadku niewykorzystywania metody decymacji optymalnym jest wykorzystanie oddzielnych kerneli, rejestrów dla przestrzeni stanów i bezpośredniego dostępu do pamięci globalnej dla danych wejściowych.

Słowa kluczowe: estymacja, śledzenie ruchu, równoległe przetwarzanie obrazów, GPGPU, sledzenie przed detekcją.

1. Introduction

Track-Before-Detect (TBD) systems are used for object (target) tracking applications when the object signal is at or below a noise floor [2-4]. Such case is very important in numerous applications because there are always limitations of sensors. Applications of TBD algorithm give abilities of a sensor range extensions and tracking under heavy conditions (e.g. fog, night, dust). Such situations are important not only in military but also in civil applications.

Conventional tracking systems are based on the detection (threshold based algorithms are used typically), the tracking (the Kalman, the Bayes, and the Benedict-Boerdner algorithms are used typically), and the assignment [2]. Track formation for particular object needs a sophisticated algorithms and a long-time verification. Selection of algorithms is especially challenging task for real-time applications. Theory of conventional tracking systems is quite well established for single object tracking systems. Multiple-object and multiple-sensor systems are especially hard to implement. TBD systems support multiple-object and they have abilities of multiple-sensor data fusion. A low signal to noise floor case complicate processing, and TBD algorithm is very computationally demanding. Single target tracking systems based on conventional techniques could be implemented at very low cost. Fixed or adaptive threshold algorithm for the detection, the Kalman filter for the tracking, and the Nearest Neighborhood algorithm for the assignment are used typically and not demanding.

TBD systems process all possible trajectories and the tracking algorithm is used for all of them independently. From practical point-of-view it is not possible to tracks all trajectories so approximations are used with a switching between similar trajectories. Such technique reduces computational requirements and switching process could be defined by the Markov's matrix.

Processing the TBD algorithms with all trajectories is necessary for high-reliability applications. Today based devices like: FPGA (Field Programmable Gate Array chips), VLSI (Very Large Scale Integration chips), GPGPU (General Purpose Graphics Processing Unit chips) are used as a computing platforms. SIMD (Single Instruction Multiple Data) processors and DSP's could also be used. TBD algorithms could be process in parallel and cluster processing is possible also.

2. Spatio-Temporal TBD algorithm

Spatio-Temporal (or Spatial-Temporal) TBD algorithm (ST TBD) is very important because could be implemented on most computing platforms at quite low cost. Overall TBD system is shown in Fig. 1 as the first part of this algorithm. The detection is for example a most probable track selection using e.g. maximal value.

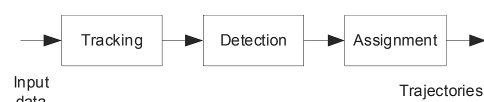

Fig. 1. Model of Track-Before-Detect system
Rys. 1. Model systemu śledzenia przed detekcją

TBD algorithm uses accumulative approach (well known denoising technique) extended by the motion model. Spatio-Temporal TBD could be implemented as a recurrent algorithm and has a following pseudocode:

Start
// initialization:
$$P(k=0,s)=0 \tag{1a}$$
For $k \geq 1$
//motion update:
$$P^{-}(k,s)=\int_{S} q_k(s \mid s_{k-1}) P(k-1,s_{k-1}) ds_{k-1} \tag{1b}$$
//information update:
$$P(k,s)=\alpha P^{-}(k,s)+(1-\alpha)X(k,s) \tag{1c}$$
EndFor
Stop

where:

k – iteration number,
s – particular space,
X – input data,
P^{-} – predicted TBD output,
P – TBD output,
α – weight (smoothing coefficient),
$q_k(s \mid s_{k-1})$ – Markov's matrix.

Such algorithm could be simplified for the linear trajectories and the constant velocity object. Such assumption reduces possibility of tracking for maneuvering object but is very simple in implementation [5]:

$$P_h = \alpha P_{h-1} + (1-\alpha) X_h \, . \quad (2)$$

Incorporating of object maneuver by Markov's matrix is simple but direct implementations are not recommended. This matrix is sparse and should be implemented directly in code not as an additional table.

3. Decimation and TBD algorithm

The output of ST TBD algorithm has increased dimensionality. The output is two-dimensional for the one-dimensional input and fixed set for the one-dimensional velocities. The output is four-dimensional for the two-dimensional input and fixed set for the two-dimensional velocities (different velocities and directions).

Output values and state-space (predicted or update) are the same and should be stored in memory. Assuming 1000x1000 input measurements at one time, and 10 motion vectors there are 10 millions of read and write operations and the memory size should store 10 millions of words (e.g. bytes, floats). Typical computing devices support small size memory with a very fast read/write access and large but slower memory. This case occurs for typical processors (general purpose, DSP, and GPGPU).

In [1] is proposed solution based on the decimation of the output. State-space necessary for iterative processing is stored inside fast memory. After a fixed number of iterations results are stored in external but slow memory. All measurements all processed like in original TBD system and only a selected output result (N'th) is delivered to the detection algorithm (Fig. 2).

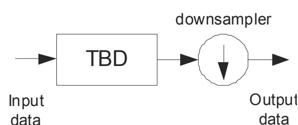

Fig. 2. Model of TBD system with decimated (downsampled) output
Rys. 2. Model systemu TBD z decymatorem

Such system could be acceptable for many applications because accumulative approach establish a low-pass filtering of input data. A high frequency component does not provide additional information and could be eliminated by the downsampler (decimator).

Input data are processed using non-overlapped window technique. Some region (selected by the window position) is processed at one time using fast internal memory as state-space storage (initialized by the previous readout of the Global Memory, used for temporal results storage and transmitted to the external memory after processing). New measurements for corresponding region are transferred from the Global Memory only as state-space updates. Such reformulation of TBD processing adds some delays to the output result but improves significantly computation speed (up to a few times for considered GPGPU [1]). Additional improvements are possible and are considered in this paper.

4. Code optimization techniques for CUDA

GPGPU code writing is possible using extended C-language. Details of GPGPU implementation and architecture constraints are manufacture's secret so without tests it is not possible to find the optimal one solution. Code profiling (manual and automatic) is necessary with some variable parameters and code implementations. This is not convenient but availability of GPGPUs is very important for today computational problems. Proper code design gives and code profiling gives ability of significant speed-up in comparison to the conventional CPUs. Typical code could be a few times faster and sometimes are large speed-ups like one hundred or more. CUDA (Compute Unified Device Architecture) is the software platform for Nvidia graphics cards and accelerators. CUDA implement parallel processing using multiple threads organized into blocks (1D, 2D, or 3D) [6,7]. Single instruction is processed by the block of threads so this is the SIMT processing (Single Instruction Multiple Threads). Blocks could be processed in parallel if there are enough processing resources.

Nvidia G80 GPU (Geforce 8800 GTS, 128 stream processors, 650MHz core clock, 1625MHz shader clock, 1944MHz memory data rate, 256-bit memory interface, PCI Express x16) is used for tests. Floating (32-bit coded) values are used for input and output data. Input measurement has 1024x1024 size and the output state space has 13x1024x1024 size so there are 13 of the motion vectors processed.

Single trajectory is assigned to every thread what gives abilities of testing different code optimization techniques. Input images are organized as a two-dimensional blocks that could be accesses directly as a memory or as a texture. The number of images depends on decimation coefficient.

The first technique gives ability of common access of multiple threads to the single memory word at one time (this is coalescence read). Achieving of coalescence accesses greatly reduce a computation time. The second technique gives ability of using additional cache memory assigned to the texture unit. There is also another approach based on temporal storage of input measurements in the Shared Memory but is not considered in this paper. Coalescence needs a synchronization of accesses of multiple threads and it is achieved by the '__syncthreads()' function call.

The next one optimization technique is the loop unrolling. This is possible by applications of '#pragma unroll level' directive before the loop (e.g. 'for' loop). The 'level' value is used for specifying of the number of loop iteration to unroll. In all examples all loops are unrolled if possible but unrolling of the loop is register consuming operation. Results could be better or not depending on code stored inside the loop and the number of iterations. Another possibility is the manual loop unrolling. There is a code for single trajectory step inside the loop, and the number of iterations is the decimation coefficient.

Application of the loop unrolling is limited because CUDA does not support unrolling if the texture access is inside in the loop.

There are some limitations of the direct memory access for boundary regions. Implementation of motion vectors could give accesses outside 2D table and what it is not recommended. Additional code for the index value control is necessary. Texture read operations support such cases without additional index control so code is smaller and could be faster. It is not guaranteed that such index control reduce overall speed because code execution is very fast and the slowest part of the system is the external memory. Only tests can be used for selection of the best solution.

CUDA support blocks processing at one time by the piece of code named as a kernel [6,7]. Single kernel could process all motion vectors at one execution time or not. The second possibility is a similar to the loop unrolling – this kernel unrolling. Kernel could be also called a few times with different parameters

(e.g. motion vectors – two float or integer values). Motion vectors could be also located in additional table.

Single kernel is called in tests with different parameters or multiple kernels are called separately. The second possibility is applied for the code improvement. Some motion vectors have 0,-1, or +1 value. During kernel compilation optimizing the CUDA compiler has a possibility of code reduction for such cases. This is not possible during run-time when the motion vector is delivered from the master function. Multiple kernels are quite rare in typical CUDA examples but are possible and reduce cost of function (kernel) call also.

The state space could be stored in the Shared Memory what is very important for fast processing and temporary results storage. This is the first option and the second is based on the register only. Single trajectory needs only one state space value for simplified formula (2). Applications of more a complex transitions using Markov's matrix is possible only by the usage of the Shared Memory, because neighborhood trajectory results should be used in calculations. All presented techniques could be mixed together for the best results.

5. Code profiling – variable parameters

Independently on the code there are some parameters related to the execution of the kernel or kernels. Kernel code could be independent on the block size. Modification of the block size influences significantly on the execution time. This is result of coalescence and cache deep if the texture unit is used. There are 8 blocks widths (8, 16, 24, 32, 40, 48, 56, and 64 of floats), 16 blocks height (1-16) and 20 downsampling values (1-20, where the 1 is related to the case without downsampling). Tests of all cases for fixed code takes about 2 hours using specially designed set of scripts for automatic compilations. Obtained results are specific to the particular GPU and memory, but most results could be extrapolated to the similar architectures.

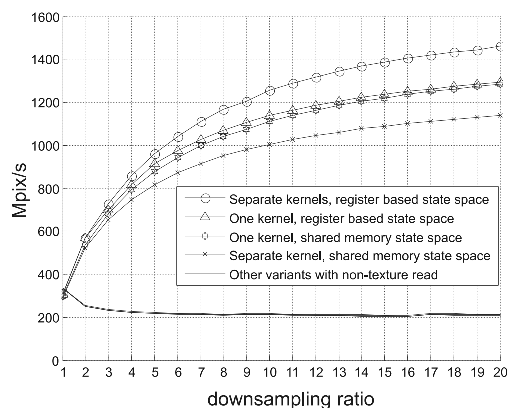

Fig. 3. Comparison of the best results for different algorithm variants
Rys. 3. Porównanie najlepszych wyników dla różnych wariantów algorytmu

Not all combinations of block sizes are possible due to limitations of GPGPU and are rejected. Every test is repeated a 10 times and the mean value is returned.

In Fig.3 are shown results for different downsampling ratios. The first group has excellent performance (a few times higher in comparison to the conventional algorithm without decimation). All algorithm implementations from this group use texture cache for input data.

The second group is related to the direct access to the external memory and has poor performance but for non-decimated processing is about 10% better what is shown in Fig.4.

For both groups the best solution is the separate kernel approach what is possible by specific code optimization by CUDA compiler. Registers used for storage of state space value is also better is comparison to the Shared Memory variant. It is possible because code has a low memory usage and there are some free registers in the processing unit.

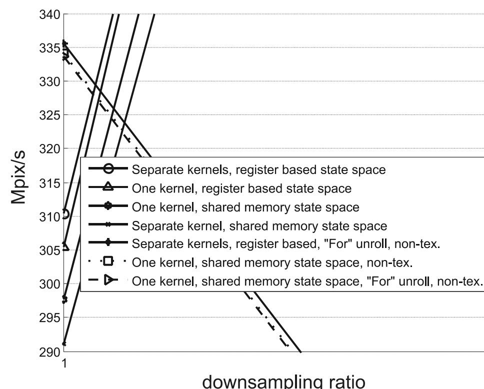

Fig. 4. Comparison of the best results for different algorithm variants
Rys. 4. Porównanie najlepszych wyników dla różnych wariantów algorytmu

The 'for' loop unroll for downsampling ratio 1 is not possible because there are nothing to unroll so this technique is useless for this code and architecture. Unfortunately the lack of unrolling for texture read is a CUDA disadvantage.

6. Conclusions

In the paper are compared two multiple implementations of TBD code variants for different parameters. The main bottlenecks of recurrent TBD algorithm implementation on GPGPU are the memory transfers. Using downsampling approach a few times higher performance could be obtained. The best solution for downsampled approach is based on the separate kernels (optimized for particular motion vectors), register based state-space (without Shared Memory) and texture cache for input measurements. Alternative approach (without downsampling) is for separate kernels, register based state-space, and direct access to measurements (without texture cache). Application of GPGPU for TBD systems is very important due to performance to cost high ratio and market's availability.

This work is supported by the MNiSW grant N514 004 32/0434 (Poland). This work is supported by the UE EFRR ZPORR project Z/2.32/I/1.3.1/267/05 "Szczecin University of Technology - Research and Education Center of Modern Multimedia Technologies" (Poland).

7. References

[1] Mazurek P.: Optimization of Bayesian Track-Before-Detect Algorithms for GPGPUs Implementations, Electrical Review R.86 7/2010, 187-189, 2010.

[2] Blackman S., Poupoli R.: Modern Tracking Systems. Artech House, 1999.

[3] Bar-Shalom Y.: Multitarget-Multisensor Tracking: Applications and Advances. Vol II, 1998.

[4] Stone L. D., Barlow C. A., Corwin T. L.: Bayesian Multiple Target Tracking. Artech House 1999.

[5] Mazurek P.: Implementation of Spatio-Temporal Track-Before-Detect Algorithm using GPU. Pomiary Automatyka Kontrola, vol. 55 nr 8, 657-659, 2009.

[6] NVIDIA CUDA - Compute Unified Device Architecture. Reference Manual v2.0, Nvidia 2008.

[7] NVIDIA CUDA - Compute Unified Device Architecture. Programming Guide v2.0, Nvidia 2008.

otrzymano / received: 10.09.2010
przyjęto do druku / accepted: 01.11.2010 *artykuł recenzowany*