Memory Efficient Deployment of an Optical Flow Algorithm on GPU using OpenMP

Olfa Haggui, Claude Tadonki, Fatma Sayadi, Bouraoui Ouni

Memory Efficient Deployment of an Optical Flow Algorithm on GPU using OpenMP

Olfa Haggui[1,2], Claude Tadonki[1], Fatma Sayadi[3], and Bouraoui Ouni[2]

[1] Centre de Recherche en Informatique (CRI), Mines ParisTech-PSL University, 60 boulevard Saint-Michel, 75006 Paris, France
[2] Networked Objects Control and Communications Systems (NOCCS), Sousse National School of Engineering, University of Sousse, BP 264 Sousse Erriadh 4023, Tunisia
[3] Electronics and Microelectronics Laboratory, Faculty of Sciences, University of Monastir, 5000 Monastir, Tunisia
{olfa.haggui,claude.tadonki}@mines-paristech.fr

Abstract. In this paper, we consider the recent set of OpenMP directives related to GPU deployment and seek an evaluation through the case of an optical flow algorithm. We start by investigating various agnostic transformations that attempt to improve memory efficiency. Our case study is the so-called *Lucas-Kanade* algorithm, which is typically composed of a series of convolution masks (approximation of the derivatives) followed by 2×2 linear systems for the optical flow vectors. Since, we are dealing with a stencil computation for each stage of the algorithm, the overhead of memory accesses together with the impact on parallel scalability are expected to be noticeable, especially with the complexity of the GPU memory system. We compare our OpenMP implementation with an OpenACC one from our previous work, both on a Quadro P5000.

Keywords: Optical flow · Lucas-Kanade · Optimization · GPU · OpenMP

1 Introduction

The use of hybride programming model has gained an increasing attention in recent years, with a special consideration on heterogeneous architectures. In order to take advantage of modern computing resources, scientific application developers need to make significant changes to their implementations. There are many benchmarks and applications in computer vision that are built with OpenMP 4.x and 5.x [1–3], which provide an excellent opportunity to get access to the noticeable computational power of the GPUs through a directive based deployment. However, this programming style generally yields some unnecessary overhead that is inherent to the paradigm, thus the programmer needs to consider this aspect beside its application driven optimizations. The main goal of the current investigation is to study how to get a more efficient implementation from a code written by an experienced OpenMP programmer with no specific GPU programming skills. To address the challenges on improving the execution

of high-level parallel code in GPU using OpenMP, we chose the Lucas-Kanade optical flow algorithm. So the aim is to derive and evaluate an optimized GPU implementation of the optical flow algorithm using OpenMP and to highlight the main programming techniques that we have considered. Implementation of the Lucas-Kanade algorithm[5] on the graphics processor Unit (GPU) is seriously considered. Regarding the multicore parallelization of the algorithm, the work by [10] for instance describes an updated method in order to speed up the objects movement between frames in a video sequence using OpenMP. Another multi-core parallelization is proposed in [11]. Pal, Biemann and Baumgartner[12] discuss how the velocity of vehicles can be estimated using optical flow implementation parallelized with OpenMP. Moreover, another hybrid model mitigate the bottleneck of motion estimation algorithms with a small percentage of source code modification. In [16], Nelson and Jorge proposed the first implementation of optical flow of Lucas-kanade algorithm based on directives of OpenACC programming paradigms on GPU. In the same context of hybride model, OpenMP provides an excellent opportunity to target hardware accelerators (GPUs) with the new version(4.0,4.5) which is very similar to the OpenACC model. In order to take advantage, many research begun using OpenMP GPU offloading in different domain. However, the implementation of optical flow algorithm with the new version of OpenMP still limited until now. In this context, this research aims to accomplish an efficient application of Lucas-Kanade algorithm for intensive computation using OpenMP GPU offloading implementation which processes and analyzes the bottlenecks of the accesses memory. In this paper, there are a number of contributions presented: First, we propose a sequential optimization strategies to improve the performance. We also evaluate the different challenges of implementing several of them to overcome some memory problems. Then, we explore the feasibility of a high level directive based model OpenMP4.0 to port Lucas-Kanade algorithm to heterogeneous architecture(GPU) using offloading model. Finally, we compare the performance obtained from a new OpenMP version and the OpenACC implementation[4] described in our previous work.

The remainder of the paper is organized as follows. Section 2 provides a basic background of the optical flow method and describes the Lucas-kanade algorithm. Our sequential optimization strategies are explained in section 3. In Section 4, we investigate the impact of GPU parallelization and optimization, we provide a commented report of our experimental results, and compare our results with the OpenACC results from our previous work. Section 5 concludes the paper and outlines some perspectives.

2 Optical Flow algorithm

Optical flow is a family of algorithms which are used to calculate the apparent motion of features across two consecutive frames of a given video, thus estimating a global parametric transformation and local deformations. It is based mainly on local spatio-temporal convolutions that are applied consecutively. The optical flow is an important clue for motion estimation, tracking, surveillance, and

recognition applications. To estimate optical flow in real time is a challenging task, it requires a lot of computation effort. So more than a hundred optical flow algorithm exist, Horn and Schunck algorithm[6] and Lucas-kanade algorithm [5] have became the most widely used techniques in computer vision. This article focuses on Lucas-kanades approach because is the most adequate in terms of calculation complexity and requires less computing resources. Its computation method is suitable for CPU and GPU implementations. The main principle of the Lucas-Kanade optical flow estimation is to assume the brightness constancy to find the velocity vector between two successive frames (t and t+1) as shown in Figure 1, (a) and (b). The optical flow vectors are drawn in Figure 1 (c).

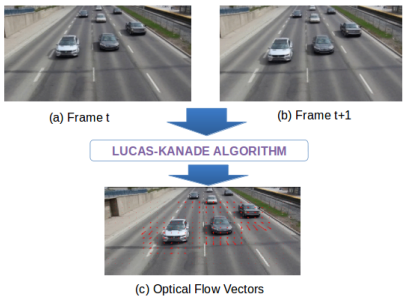

Fig. 1: Optical Flow computation

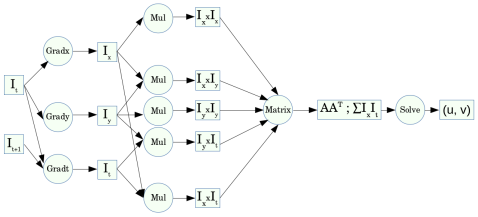

Fig. 2: Workflow of Lucas-Kanade

2.1 Lucas-Kanade algorithm

The idea of Lucas-Kanade is to compute the spatiotemporal derivatives on a smoothed image to minimize the intensity variations over time. This requires a focus on a representative pixels which are then checked for motion across consecutive frames through intensity variations between the scene and the camera, followed by a construction of the least square matrix in a spatial neighborhood to calculate the optical velocity flow. Consider for a 2D image I, a small motion is approximated by a translation. We need to determine the motion flow vectors for the image pixel $I(x, y)$. Thus if the current frame is represented by its intensity function I, then the intensity function H of the next frame is such that where (u, v) is the displacement vector. Here, we briefly describe the correspondence of equations with different steps of the Lucas-Kanade algorithm.

$$H(x, y) = I(x + u, y + v), \tag{1}$$

Therefore, we have to solve for every pixel the following so-called *Lucas-Kanade equation*:

$$\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \tag{2}$$

where I_x, I_y and I_t are the derivatives of the intensity along x, y and t direction respectively. A least-square approach are implemented in Lucas-Kanade system to find the most likely displacement (u, v), since the original system is over determined. The summations within equation (2) are over the pixels inside the sampling window. If the condition number of the normal matrix is above a given threshold, then we compute the solution of the system (using Kramer method for instance) and thus obtain the components of the optical flow vector for the corresponding pixel. Figure 2 summarizes a sequence of computation stages of the Lucas-Kanade algorithm where the derivatives I_x, I_y, and I_t are computed through their Taylor approximations using the corresponding convolution 6 kernels. Then follows their point-wise products compute the products I_x^2, I_y^2, I_xI_t, I_yI_t, and I_xI_y. Computes for each pixel the normal matrix and the right hand side of the linear system as described in equation (2).

3 Sequential optimization strategies

For many algorithms, especially in computer vision and image processing field, a stencil computation are a common programming pattern. Usually, image processing algorithms combine the challenges of stencil computations and that of real-time processing. Therefore, an efficient implementation requires to focus on data locality and to consider a scalable parallelisation. More precisely, We should take care about redundant memory accesses, cache misses, and unalignement issues. We now describe some techniques that we have considered for the aforementioned concerns.

3.1 Operators Clustering

Operators clustering aims at merging two or more operators into a single one, in order to reduce the lifetime of the intermediate results in between and to improve data locality. In its *Nopipe* version, the Lucas-Kanade algorithm is composed of four computation stages: computation of the gradients, product of the gradients, computation of the matrix coefficients together with the corresponding right and sides, and solving the linear systems for the optical flow vectors. This computation chain requires accessing nine intermediate arrays. For our case, several combinations are possible [15]. For instance, we can choose to pipeline the Grad and Mul operators one hand, and the Matrix and solve operators on the other hand, this transformation is called *Halfpipe* and it reduces both the number of floating point operations and memory accesses. For our scenario, the most balanced one seems to fully pipeline the operators, thus removing all intermedi 6 ate memory accesses. This is called *full-pipe*, where we form the unique cluster GRAD+MUL+MATRICE+SOLVE. Figure 4 illustrates the full-pipe workflow.

3.2 Loop optimization

Stencils represent a challenging computational pattern for Memory optimization, the resulting computation loop therefore needs to be optimized. Loops optimiza-

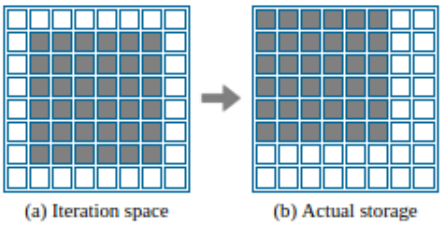

Fig. 3: Upper-left shift storage

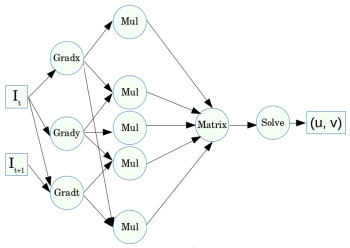

Fig. 4: Full-pipe Organization

tion plays an important role in high performance computing. Possible goals are: improving data reuse and data locality, reducing the overheads associated with loops management, and maximizing parallelism. Loop transformations can be performed at different levels for our work. In this context, we present a loop shifting technique, which minimizes the memory needed to carry on the main arrays. In fact, we apply an upper-left shift for the output matrix, which means that (i, j) is stored at position $(i-1, j-1)$ [15]. Figure 3 illustrates our re-indexation and the corresponding storage strategy. After applying loop shifting , we apply a Loop fusion to improve the readability of the code and to make programs faster by replacing multiple loops with a single one. Furthermore, loop fusion can help to have a more coarse grained parallelism. For the next point, we study the effect of array contraction on data reuse and data locality.

3.3 Array Contraction

In this paper, we consider another optimization strategy to benefit from the cache. The so-called *array contraction*, which aims at reducing the memory footprint [14], is a program transformation which reduces the size of intermediate arrays by means of location reuses. We use the modulo to round up on i direction. This approach clearly reduce the number of loads. In order to improve the register use, we consider a special case of array contraction, namely *scalarization*, where each element of an array is defined and immediately used within the same iteration. To illustrate the potential benefit of these strategies, we include preliminary results showing the improvement that is achieved when collective loop transformations with a contraction array are applied. We run on a dual-socket intel broadwell. Table 1 shows the performance results of our strategy using different image sizes. We can see that with the optimized case we achieve better execution times compared to the basic one, and we can also process larger images (like 8000x8000 and 16000x16000), which was not possible with the basic implementation because of memory limitations. Figure 5 shows a noticeable improvement, which demonstrates the impact of our optimization and the potential of 6 more performance gain.

Table 1: Evaluation of the sequential optimization

image size	T(s) Basic	T(s) Optimized	% Improvement
2000^2	0.752	0.078	89.62
4000^2	1.058	0.315	70.22
8000^2	1.775	1.264	28.79
12000^2	-	2.846	-
16000^2	-	5.065	-

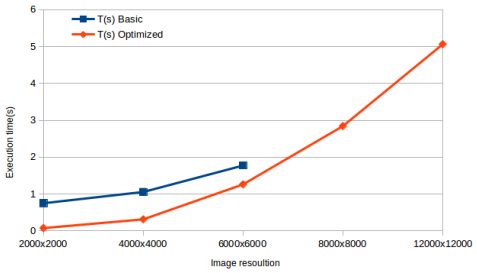

Fig. 5: Improvement of sequential implementation

4 Impact of GPU Parallelization and Optimization

4.1 Hardware Configuration

We use OpenMP4.0 and run on an NVIDIA Quadro P5000 GPU accelerator (Pascal architecture). It includes 2560 CUDA cores with 16 GB GDDR5 memory. The host is an Intel(R) Xeon(R) CPU E5-1620 v4 processors with 4 cores. We use GCC compiler version 7.3.

4.2 Results of the GPU Parallelization

Our performance analysis considers different stages, each focusing on the comparison between the results obtained with our OpenMP4.0 implementation and those of our previous OpenACC implementation. Some directive based optimization were performed to improve memory accesses and memory locality. In our experiments, we use different frame sizes and run our Lucas-Kanade implementation for the optical flow vectors. We start with a baseline sequential implementation in C, then we consider the derived OpenMP version without any data directives. OpenMP4.0 version introduces a number of features for targeting heterogeneous architectures [13]. The first step of the algorithm consists in loading the data from the CPU to the GPUs global memory. This step (typically) yields a significant overhead. Then, we define which part will be accelerated with the device (kernel) using the basic directives (`#pragma omp target`). The target directives provide a mechanism to move the execution of the thread from the CPU to another device and to relocate the data. We can see from table 3 that the parallel CPU version outperforms the OpenMP code on the GPU. The reason of this is that OpenMP was originally designed for automatic multithreading on a shared memory processors, so the parallel directive only creates a single level of parallelism. Beside this one, the version at this stage does not contain any optimization directive, so there is a *potential* room for improvement. To evaluate this potential, we used the NVIDIA runtime profiler on our kernels to identify locations where memory access seems too important. We use `nvprof`

`--print-gpu-trace` to print all the information about how to optimize the code. We can see from the results another clear difference between OpenMP and OpenACC implementations, where the performance achieved by the OpenACC is quite higher compared to OpenMP and CPU versions. However, OpenACC is very similar to OpenMP but OpenACC was designed from the beginning to address both portability and productivity of GPU programming without too much effort, with OpenMP we must skillfully exploit these new features. Figure 7 demonstrates the performance of GPU implementation using OpenACC. To overcome theses issues, we consider several specific directives. We start with the teams directive to express a second level of scalable parallelism. In order to make better use of GPU resources, we have used many use many thread teams via the `teams` directive and the `directive` to distribute the iterations of the next loop to the master threads of the teams in order to spread parallelism across the entire GPU. In order to increase the parallelism, we start by increasing the number of teams using the `num−teams` clause and the `num−thread` clause to generate the number of threads per teams which might yield the best performance and achieves a good balance between teams and threads. In addition, we can further increase parallelism by using our distributed and work shared parallelism from the same loop. We can collapse them together and we split `teams distribute` from `Parallel For` by moving them to the inner loop. In our case we use the `COLLAPSE(N)` clause to have the next N loops collapsed into one loop with a larger iteration space. This will give us more parallelism to distribute.

Table 2: Evaluation of the OpenMP GPU Deployment

	2000^2	4000^2	6000^2	8000^2	12000^2
(1)	0.046	0.184	0.416	0.741	1.666
(2)	0.142	0.343	0.662	0.910	2.556
(3)	0.103	0.240	0.428	0.690	1.356
(4)	0.079	0.123	0.355	0.502	1.008
(5)	0.075	0.115	0.330	0.488	0.984
(6)	0.062	0.094	0.275	0.428	0.882

Table 3: Evaluation of the GPU parallelization

Image size	CPU(s)	GPU(s) OpenMP	GPU(s) OpenACC
2000^2	0.046	0.142	0.018
4000^2	0.184	0.343	0.057
6000^2	0.416	0.628	0.131
8000^2	0.741	0.391	0.250
12000^2	1.666	2.556	0.640

- (1) : Basic CPU (2): GPU threaded
- (3) : GPU teams/distribute (4): (3) + GPU team/thread balance
- (5) : (4) + GPU collapse (6): (5) + GPU SIMD

Table 2 lists the different stages of the results, which demonstrate that there is a significant improvement most of the time. We investigate the effects on the performances when combined parallelization and vectorization in GPU. In fact, vectorization using SIMD constructs is very efficient in exploring data level parallelism since it executes multiple data operations concurrently using a single

instruction. Therefore, OpenMP4.0 provides the `pragma omp simd` directive, which execute multiple iterations of the loop using vector instructions when possible. As shown in figure 6, the highest performance we have achieved when

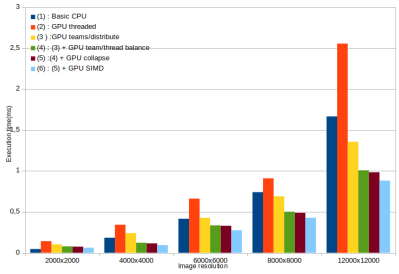

Fig. 6: Performance comparison between different OpenMP GPU offloading optimization

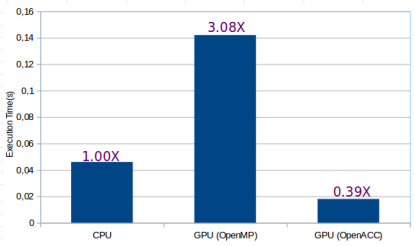

Fig. 7: The performance of GPU Offloading with OpenMP4 and OpenACC

combined all the directives used in parallelization with the vectorization, which make the results more fast but the time is still limited due to the heavy implicit access to memory .

4.3 Managing memory and data optimization

One of the major bottleneck of a GPU is the data transfer latency, because it takes more than 400-600 cycles to access the global memory. If access to global memory is frequent, then with the cost of moving data between the CPU and GPU at every loop, the computation benefit of porting a parallel application to a GPU will be lost. We discuss now how we schedule memory accesses in order to reduce the overhead of data exchanges and get ride of intermediate data accesses whenever possible. We analyze the behavior of the major data movement with the OpenMP4.0 offloading target using the target data *directives* and *map clauses* to control and reduce data movement between CPU and GPU. OpenMP4.0 allows an explicit control of data allocation together with the corresponding transactions through appropriates clauses (`copyin, copyout, present, create`) with different map-type (`to,from, tofrom,alloc`) to optimize the mapping of buffers to the device data environment .

Basically, CPUs and GPUs have separate memories and 6 can not access each other's memory which must be handled explicitly by programmers. Instead of this, a new concept of *unified memory* provided by NVIDIA, which allows the GPU and the host CPU to share the same global address space. This permits the host to refer to memory locations on the attached devices, and the devices to access addresses on their host. The unified memory enables fast memory accesses with large data sets where the movement data is managed by the underlying system automatically [1]. There is no need for address translation, and both CPU

and GPU can use the same pointer. Moreover, leveraging the unified memory features makes it feasible to run kernels with memory footprints larger than the GPU memory capacity. In current OpenMP GPU offloading , we illustrate how the function and data transfer work with the unified memory [1]. To do this, we use `omp-target-alloc` for data allocation and `is-device-ptr` clauses to pass them to target regions, in contrary to OpenACC compiler which provides the flag `-ta=tesla:managed` for the unified memory consideration. When this option is used at compile time, the PGI compilers will intercept and replace all user defined allocations with managed data allocations.

6

Table 4: Evaluation of our data GPU optimization

	2000^2	4000^2	6000^2	8000^2	12000^2
(1)	0.062	0.094	0.275	0.428	0.882
(2)	0.011	0.024	0.101	0.228	0.689
(3)	0.007	0.014	0.077	0.105	0.481
(4)	0.005	0.010	0.041	0.068	0.137

- (1): Basic GPU parallelization
- (2): (1) + Data movement performance
- (3): (1) + Unified memory
- (4): (2) + pinned

We also highlighted the benefits of using pinned me 6 mory on the memory copies which offers the best performances. Furthermore, if the memory is going to be used for many asynchronous transfers, then we request page-locked memory allocations (pinned memory). It is a memory allocated using the `cudaMallocHost` function, which prevents the memory from being swapped out and thereby provides improved transfer speeds, contrary to the non-pinned memory obtained with a plain `malloc`. The benefit of using pinned memory is that we can solve larger problems because the size of the pinned memory is much larger than that of the global memory. The pointer association between CPU-GPU is preserved, thus preventing the operating system from moving this memory to another location. Presently, none of the pragmas can allocate pinned memory and compiler flag. Hence, OpenMP GPU offloading use the `cudaMallocHost` function for pinned memory, which is not the case with OpenACC who considers the flag compiler `-ta=tesla:pinned`. The experimental results of our optimization investigation are summarized in Table 4. The figure 8 outlines the speedup ratio of the total execution time for each of the different versions compared to the baseline version. We can see a remarkable speedup with our incremental OpenMP GPU offloading data optimization, when using a simple data movement with unified memory and pinned memory directives. We have found that pinning the same amount of memory was more faster than the use of unified memory and the basic data movement directives. This technique is more efficient and it often reduces the overall amount of host-device data transfers. Overall, our work makes several important research contributions, we evaluate the effectiveness of OpenMP GPU offloading directives as a potential solution to the performance

portability problem of modern architectures and we get decent speedups. Right now, OpenACC support is very selective and limited both for devices and compilers. Whereas, OpenMP GPU offloading is very widely supported and exhaustive. However, to better understand the computational costs of the different versions,

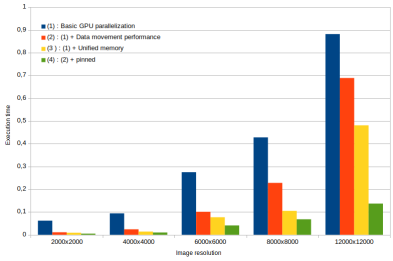

Fig. 8: Speedup of the three versions over the baseline version

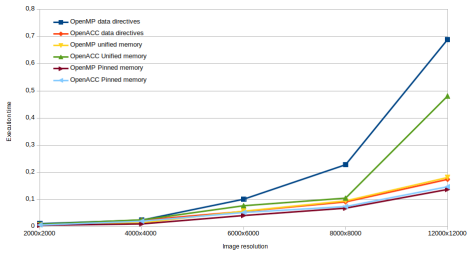

Fig. 9: Basic CPU and fully optimized GPU

figure 9 displays a comparison between the OpenMP 4.0 implementation and the OpenACC implementation for each version. As we can see, the management of the data using directives with OpenMP on the GPU is more costly than with OpenACC in this stage. However, with the use of the unified memory, we notice that the OpenMP version is better than the OpenACC version due to the use of a compiler flag. Moreover, the line labeled OpenMP Pinned Memory shows the timings of both OpenMP and OpenACC which are so close with pinned memory.

5 Conclusion

In this paper, we have carried out a detailed study of some of the most popular parallelization approaches and programming languages used to program GPUs. An OpenMP GPU offloading and OpenACC are used in this work, which are considered as the most flexible high level languages for GPU deployment. It makes possible to migrate standard CPU code in a straightforward way without making too many modifications, and obtain a decent performance compared to other complex programming models like CUDA and OpenCL. OpenMP 4.x directives provide an excellent opportunity to GPU deployment. We investigate an OpenMP deployment of the Lucas-Kanade optical flow algorithm with OpenMP and strive to obtain better performance than that of a parallel version on a manycore processor. The performances are very close to our previous OpenACC version. For the future works, we will investigate a combination of OpenMP and OpenACC both for GPU and manycore deployments .

References

1. A.Mishra , L.Li , M.Kong : Benchmarking and Evaluating Unified Memory for OpenMP GPU Offloading ,Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, ACM,New York, NY, USA,2017.
2. A. Chikin, T. Gobran, J.N. Amaral, : OpenMP Code Offloading: Splitting GPU Kernels, Pipelining Communication and Computation, and Selecting Better Grid Geometries ,5th International Workshop, WACCPD 2018, Dallas, TX, USA, November 11-17, 2018, Proceedings.
3. M. Martineau, S. McIntosh-Smith,C. Bertolli, A. C. Jacob, S. F. Antao,A. Eichenberger, G.T. Bercea,T. Chen, T. Jin, K.Brien, G. Rokos,H. Sung, Z. Sura : Performance Analysis and Optimization of Clangs OpenMP 4.5 GPU Support, 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, 2016 IEEE.
4. O.Haggui, C.Tadonki, F.Sayadi, O.Bouraoui, : Efficient GPU Implementation of Lucas-Kanade through OpenACC ,In Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 5: VISAPP, 768-775, 2019, Prague, Czech Republic.
5. B.D. Lucas, T. Kanade, : An Image Registration Technique with an Application to Stereo Vision, in Proceedings of Image Understanding Workshop, 1981, pp. 121-130.
6. B.K.P. Horn, B.G. Schunck, : Determining optical flow Artificial Intelligence, vol.17(185), pp.185203, 1981.
7. J. Gibson, : The Perception of the Visual World, Houghton Mifflin, Boston, 1950. 6
8. Siman Baker, Iain Mattews, :Lucas Kanade 20 Years On:A Unifying Framework, International Journal of Computer Vision 56(3), 221255, 2004.
9. Mancia Anguita, Javier Daz, Eduardo Ros, and F. Javier Fernandez-Baldomero, : Optimization Strategies for High-Performance Computing of Optical-Flow in General-Purpose Processors, IEEE Transactions on circuits and Systems for Video Technology, Vol. 19, NO. 10, October 2009. 6
10. A . V. Kruglov and V. N. Kruglov, : Tracking of Fast Moving Objects in Real Time , ISSN 1054-6618, Pattern Recognition and Image Analysis, 2016,Vol. 26, No. 3, pp. 582586.Pleiades Publishing, Ltd., 2016.
11. Avier Anchez, Nelson Monz, August in Salgado, : Parallel Implementation of a Robust Optical Flow Technique, Las Palmas de Gran Canaria, 16 March 2012.
12. Istvn Pl, Ronny Biemann, Stefan V. Baumgartner,:A Comparison and Validation Approach for Traffic Data, Acquired by Airborne Radar and Optical Sensors using Parallelized Lucas-Kanade Algorithm , VDE VERLAG GMBH Berlin Offenbach, Germany,ISBN 978-3-8007-3607-2 / ISSN 2197-4403,2014.
13. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf .
14. Yonghong Song, Rong Xu, Cheng Wang, Zhiyuan Li .: Improving Data Locality by Array Contraction,IEEE Transactions on Computers,2004.
15. O. Haggui, C. Tadonki, L. Lacassagne, F. Sayadi, B. Ouni, : Harris Corner Detection on a NUMA Manycore, Future Generation Computer Systems https://doi.org/DOI: 10.1016/j.future.2018.01.048, 2018.
16. Nelson Martin, Jorge Collado, Guillermo Botella, Carlos Garcia, Manuel Prieto, : OpenACC-based GPU Acceleration of an Optical Flow Algorithm,ACM Digital Library,SAC15 April 13-17, 2015, Salamanca, Spain.