GPU Implementation of an Anisotropic Huber-L^1 Dense Optical Flow Algorithm Using OpenCL

Duygu Büyükaydın, Toygar Akgün
ASELSAN
Transportation, Security, Energy and Automation Systems (UGES) Business Sector
Ankara, Turkey
Email: {dcelebi,takgun}@aselsan.com.tr

***Abstract*—Optical flow estimation aims at inferring a dense pixel-wise correspondence field between two images or video frames. It is commonly used in video processing and computer vision applications, including motion-compensated frame processing, extracting temporal features, computing stereo disparity, understanding scene context/dynamics and understanding behavior. Dense optical flow estimation is a computationally complex problem. Fortunately, a wide range of optical flow estimation algorithms are embarrassingly parallel and can efficiently be accelerated on GPUs. In this work we discuss a massively multi-threaded GPU implementation of the anisotropic Huber-L^1 optical flow estimation algorithm using OpenCL framework, which achieves per frame execution time speed-up factors up to almost 300×. Overall algorithm flow, GPU specific implementation details and performance results are presented.**

I. Introduction

Optical flow estimation aims at inferring a dense pixel-wise correspondence field between two images or video frames. Initiated by the seminal works of Lukas-Kanade [1] and Horn-Schunck [2], a wide range of optical flow estimation techniques have been developed [3] and optical flow estimation is still an active research topic.

Optical flow estimation finds widespread application in video processing, computer vision, navigation, surveillance and medicine. Most of these fields have stringent timing requirements, resulting in a high demand for fast (real-time or near real-time) implementations. However, dense optical flow estimation is an inherently computationally complex task that may also require high memory bandwidth depending on the applied algorithm. In order to meet the desired performance level substantial floating point processing power and high memory bandwidth is required. GPUs (Graphics Processing Units) are widely used for accelerating computationally demanding algorithms given the algorithm allows for a data parallel implementation [4]. GPUs can be programmed using the OpenCL (Open Computing Language) framework which supports data and task level parallelism on heterogeneous multi and many-core processors [5].

In this work we discuss a massively multi-threaded GPU implementation of the anisotropic Huber-L^1 optical flow estimation algorithm proposed in [6] using the OpenCL framework. Section II briefly describes the overall algorithm flow. Section III discusses GPU specific implementation details. Section IV presents performance results, and finally Section V concludes the paper.

II. Dense Optical Flow Algorithm

Based on a detailed literature survey, the anisotropic Huber-L^1 optical flow estimation algorithm proposed in [6] was selected as the best compromise of estimated flow field quality, computational resource requirements and suitability for parallel implementation. According to Middlebury benchmark site [7], at the time of submission of this paper, average ranking of the anisotropic Huber-L^1 algorithm is 51.1 with respect to endpoint error [8]. On the other hand, it is one of the fastest algorithms with an average per frame execution time of 2 seconds for 640×480 input frames. Furthermore, the algorithm allows for a massively multi-threaded implementation that can effectively utilize the computational resources of GPUs.

The starting point of the algorithm in [6] is finding a solution to a disparity preserving spatially continuous formulation of the optical flow problem based on L^1 data term and isotropic TV regularization. For two input images I_0 and I_1 defined on a rectangular domain $\Omega \in \mathbb{R}^2$, the cost funtion to be minimized over the optical flow vectors is stated as

$$\min_{\vec{u}} \left\{ \int_\Omega \sum_{d=1}^{2} |\nabla u_d| + \lambda |I_1(\vec{x} + \vec{u}(\vec{x})) - I_0(\vec{x})| d\vec{x} \right\}, \quad (1)$$

where $\vec{x} = (x_1, x_2)$ is the $2D$ vector of pixel coordinates, and $\vec{u}(\vec{x}) = (u_1(\vec{x}), u_2(\vec{x}))$ is the $2D$ vector of displacements (flow vectors). Free parameter λ is used to balance the relative weight of data and regularization terms. Note that data term is simply the L^1 distance between $I_1(\vec{x} + \vec{u}(\vec{x}))$ (motion warped I_1) and I_0, and regularization term is the sum of absolute gradients of the flow vectors (which is the idea behind total variation regularization). In order to simplify the minimization, an auxiliary variable $\vec{v}$ and a coupling term to ensure that $\vec{v}$ is a close approximation of $\vec{u}$ is introduced. After applying anisotropic Huber regularization and with other mathematical manipulations, details of which can be found in the original paper, Eq. (1) yields

$$\min_{\vec{u},\vec{v}} \sup_{|\vec{p}_d| \le 1} \left\{ \int_\Omega \sum_{d=1}^{2} \left[(D^{\frac{1}{2}} \nabla u_d) \cdot \vec{p}_d - \epsilon \frac{|\vec{p}_d|^2}{2} + \frac{1}{2\theta}(u_d - v_d)^2 \right] + \lambda |\rho(\vec{v}(\vec{x}))| d\vec{x} \right\}. \quad (2)$$

The regularization term that leads to the final form in Eq. (2) is stated to be *anisotropic Huber* in the sense that it is discontinuity preserving and image-driven, and uses Huber cost, which is quadratic for small differences and linear for large differences. This optimization problem is solved iteratively by an alternating minimization procedure. For fixed $\vec{v}$ solution yields to:

$$\begin{aligned} u_d^{n+1} &= v_d^n + \theta div(D^{1/2}\vec{p}_d^{n+1}), && (3) \\ \vec{p}_d^{n+1} &= \frac{\vec{p}_d^n + \tau(D^{1/2}\nabla \vec{u}_d^{n+1} - \epsilon \vec{p}_d^n)}{\max\left\{1, |\vec{p}_d^n + \tau(D^{1/2}\nabla u_d^{n+1} - \varepsilon \vec{p}_d^n)|\right\}}. && (4) \end{aligned}$$

For fixed $\vec{u}$ solution yields to:

$$\min_{\vec{v}} \left\{ \int_\Omega \frac{1}{2\theta} \sum_{d=1}^{2} (u_d - v_d)^2 + \lambda |\rho(\vec{v}(\vec{x}))| d\vec{x} \right\}. \qquad (5)$$

A thresholding step with three different cases yields a direct solution:

$$\vec{v}^{n+1} = \vec{u}^{n+1} + \begin{cases} \lambda\theta\nabla I_1, \text{ if } \rho(\vec{u}^{n+1}) < -\lambda\theta|\nabla I_1|^2 \\ -\lambda\theta\nabla I_1, \text{ if } \rho(\vec{u}^{n+1}) > -\lambda\theta|\nabla I_1|^2 \\ -\rho(\vec{u}^{n+1})\dfrac{\nabla I_1}{|\nabla I_1|^2}, \text{ else }. \end{cases} \qquad (6)$$

Eq. (3), (4) and (6) present the overall solution to be implemented by replacing all mathematical operations with their discreet approximations that can be computed on rectangular image grids.

III. IMPLEMENTATION

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs (Central Processing Units), GPUs (Graphic Processing Units), DSPs (Digital Signal Processors), FPGAs (Field Programmable Gate Arrays) and other processors [5]. OpenCL supports data and task level parallelism through a simple to understand and use programming model. Furthermore, since it supports different architectures OpenCL provides portability to a great extent.

Since the programming model of OpenCL is closely related to multi/many-core vector processors, it makes good sense to describe it on an example. The GPUs that are used for testing the OpenCL implementation reported in this work are the NVIDIA GTX-780 and GT-640. Based on the full Kepler GK110 chip implementation, GTX-780 has 12 active (out of 15) SMX (streaming multi-processor) units, corresponding to 2304 (192 $\times$ 12) CUDA cores [9], [10]. Its theoretical peak floating point performance is 3977 GFLOPs. Based on the Kepler GK107 chip (scaled down GK110), GT640 has 2 active SMX units, corresponding to 384 (192 $\times$ 2) CUDA cores. Its theoretical peak floating point performance is 663 GFLOPs. The overall architecture of GK110 is shown in Figure 1. Each of the GK110 SMX units, as shown in Figure 2, has 192 CUDA cores, and each core has fully pipelined floating-point (IEEE 754-2008 compliant single and double precision with fused multiply-add operation) and integer arithmetic logic units. Both GTX-780 and GT-640 support OpenCL 1.2. OpenCL programming model allows the user to define global and local work sizes. Global work size is the total number of threads to be executed. Assuming the thread to data point mapping is designed in such way that each thread processes one data point (also assuming input and output data rates are equal for simplicity), the global work size is the total number of threads

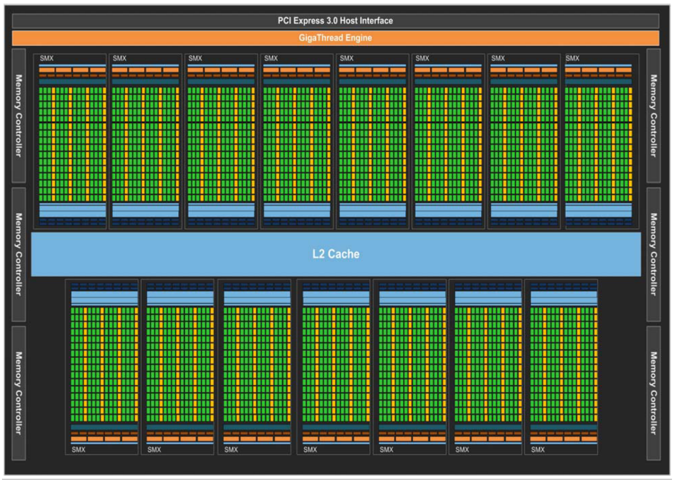

Fig. 1: Full GK110 chip with 15 SMXs

to be executed, which is also equal to the total number of data points (pixels, in our case) to be processed. On the other hand, local work size is the total number of threads that belong in a work-group. Threads in a work-group can be synchronized by local and global memory fences and can efficiently exchange information via local memory, which typically corresponds to L1 cache. The idea behind work-groups is that a work-

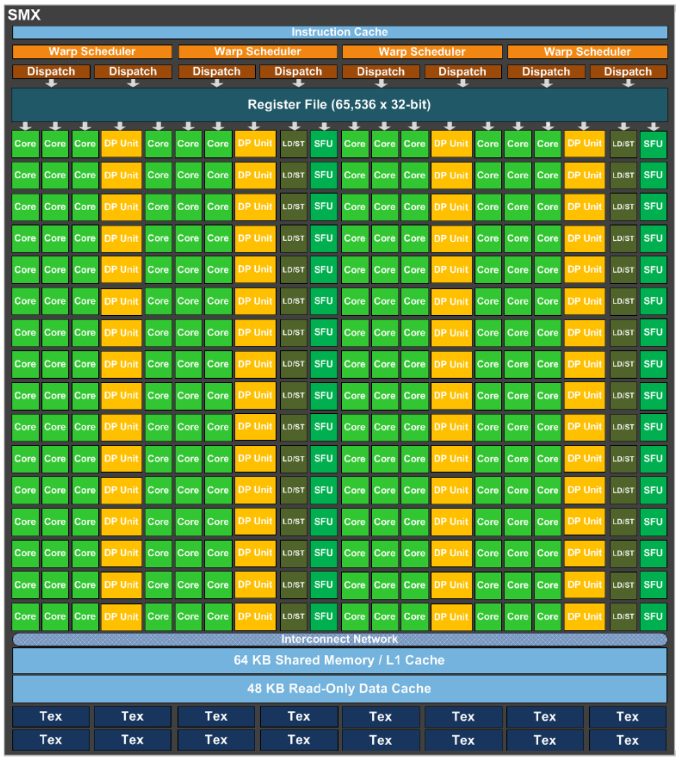
SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Fig. 2: Kepler SMX with 192 cores

group corresponds to a chunk of threads that can execute independent of the rest of the global work. Using our example GPUs, each work-group is assigned to an SMX (Fig. 1) for execution and each thread within a work-group is executed on

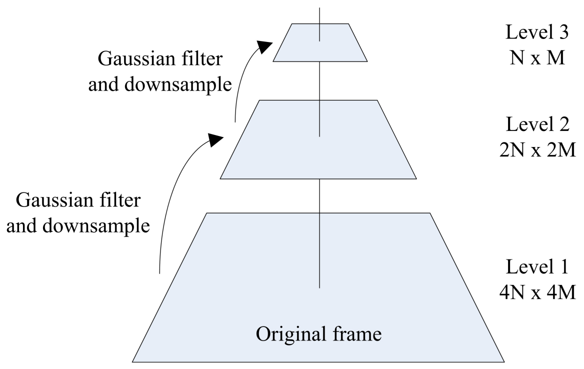

Fig. 3: Image pyramid

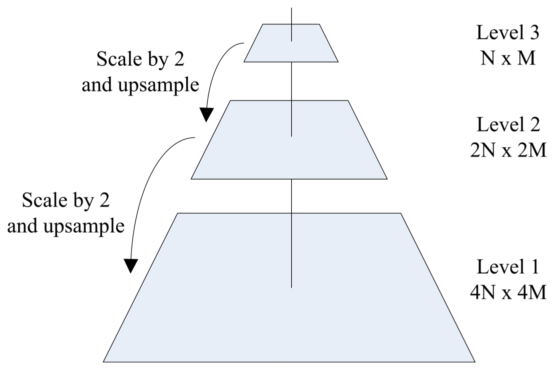

Fig. 4: Flow vector pyramids

one of 192 CUDA cores (Fig. 2) on that SMX. Since work-groups do not wait for each other, the work-group to SMX assignment is completely independent of the work-group order, completion of a given work-group or the number of available SMXs, providing scalability. This property will be clearly demonstrated by our test results, where identical OpenCL implementations (due to identical SMX structures) running on GTX-780 and GT-640 will have scaled performance figures. Note that the expected performance scaling is around 6$\times$ (2304 / 384) since the algorithm under inspection consists of mostly compute bound (number of operations per memory access is high) kernels.

For our implementation, we started with the MATLAB source code that is given as a reference in the original paper which can be downloaded from [11]. MATLAB source was rewritten in C/OpenCL where simple setup and frame capture functionality was implemented in C and flow estimation related functions were implemented directly in OpenCL and ported to GPU. The mainframe of the implementations is based on a classical pyramidal optical flow solution method. Initially, pyramids of the two input images I_0 and I_1 are built in which resolution decreases as level of the pyramid increases. The pyramid structure for an input image is shown in Figure 3. The image pyramids are built by applying a Gaussian low pass filter followed by downsampling. Low pass filtering prior to downsampling is required to avoid aliasing artifacts. The details of the filter implementation is given in the following section. Then the algorithm runs from the top level to the lowest level using the resulting optical flow vector from one level up. The implementation requires four major surfaces:

- u: optical flow vector's x direction component, has the same size as the input frame
- v: optical flow vector's y direction component, has the same size as the input frame
- w: auxiliary variable, has the same size as the input frame
- p: auxiliary variable that stores x and y gradients for u, v and w surfaces, has six times the size of the input frame

The execution steps in our pyramidal implementation are based on these variables and images The overall execution flow is as follows:

1) Initialize u, v, w, p zero. Run the algorithm at the third level, which is the top level of the pyramid, to obtain updated surfaces.
2) Upsample u, v, w, p by 2 and scale them by 2.
3) Run the the algorithm at second level with the scaled and upsampled optical flow vector and auxiliary surfaces to obtain updated surfaces.
4) Upsample u, v, w, p by 2 and scale them by 2.
5) Run the algorithm at third level with the scaled and upsampled optical flow vector and auxiliary surfaces to obtain updated surfaces.
6) Report the output optical flow vector surface as the result.

Note that motion vectors are scaled by 2 to compensate for changing resolution between pyramid levels. The flow vector pyramids are shown in Figure 4. The execution flow in each pyramid level is shown in Figure 5.

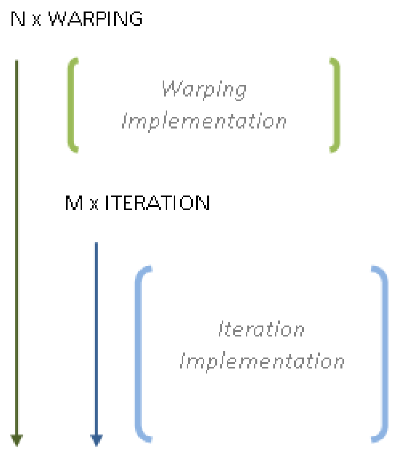

Fig. 5: Algorithm flow

A. Gaussian Filtering

The Gaussian filter that is used to produce the images at upper levels of pyramid is a 5×5 2D separable filter. In order to reduce computational complexity, we implemented this 2D filtering operation as two consecutive 1D filtering operations; first horizontal then vertical. The filter weights are $[1, 4, 6, 4, 1]/16$ in both directions. In these kernels local memory buffer is used to increase the efficiency since in filtering operation neighboring data points are used. In OpenCL context, local memory can be interpreted as user controlled L1 cache where the user is responsible for designing the thread mapping for reading pixel values from DRAM and writing into the L1 memory. For both kernels $2D$ thread arrays are used with a local work size of 64×4 (in horizontal and vertical directions respectively).

B. Warping

The term image warping means that each pixel of the source image mapped with its flow vector and written into destination image [12]. Initially, texture units were considered to implement warping. Available only for GPUs, texture hardware inherently implements nearest neighborhood and bilinear interpolation and provides 2D access optimized caching mechanism for efficient reuse of neighboring elements. However, texture units operate on OpenCL image buffers, resulting in additional surface copies from regular OpenCL buffers. As a result, the final implementation uses regular OpenCL buffers and implements the $2D$ interpolation calculations manually. The warping kernel is implemented with a single kernel that uses a $2D$ thread array with a local work size of 64×4.

C. Iterations

After each warping step, optical flow iterations are executed on the warped surfaces. An iteration is basically composed of four steps:

1) Update p
2) Normalize p
3) Update u, v, w
4) Thresholding

For these surfaces, separate memory regions are allocated at each pyramid level. Using same memory region for every level of pyramid is much slower compared to using separate memory regions. Moreover extra memory requirement is not high with respect to total available global memory.

Software is configured in such a way that first and second steps are implemented as one kernel and last two steps are implemented as another kernel. Initially, all four steps were merged into a single kernel. However, the last two steps (3,4) cannot start execution before the first two steps (1,2) execute to completion. This results in a local memory fence in the middle of the merged kernel. Furthermore, the merged kernel requires a larger number of resources (especially registers) to execute. In the light of these observations and performance tests conducted with the merged kernel, the final implementation was based on the separated design that implements the first two steps as one kernel (updateAndNormalizeP) and the last two steps as another (updateFlowVectors).

1) Kernel updateAndNormalizeP: In this kernel p is updated using derivatives of u, v, w and then p is normalized so that every element's absolute value is less than one.

Derivative operation can benefit from local memory buffer (threads processing neighboring pixels need to read overlapping pixels). Therefore we implemented this kernel using local memory cache. For efficient DRAM bandwidth use reads and writes were done as 4 wide float vectors. The vector data type used in our application is `float4` that is composed of four floating point pixels. The second part of this kernel is normalization which is not suitable for any major optimization, except for DRAM bandwidth optimization which is mostly taken care of by `float4` read/writes. Each data point is checked separately and normalized if necessary. The updateAndNormalizeP kernel uses a $2D$ thread array with a local work size of 32×4.

2) updateFlowVectors: This kernel has two steps:

- Updating u, v, w using derivative of p
- Thresholding

This kernel is similar to updateAndNormalizeP kernel but it is slightly more complex. In thresholding step, there are two thresholds to compare. Each data point is checked separately and necessary calculations are made. The updateFlowVectors kernel uses a $2D$ thread array with a local work size of 32×4.

D. Assisting Kernels

Apart from these kernels that perform most of the calculations, there are some assisting kernels. These simple kernels do not contribute to the main algorithm but simply assist the execution flow:

- copyKernel: Copies one surface to another. Uses a $2D$ thread array with a local work size of 32×4. All reads and writes are performed in terms of `float4`.
- fillWithZerosKernel: Fills a surface with zeros. Uses a $2D$ thread array with a local work size of 32×4. All writes are performed in terms of `float4`.
- upsampleAndScale: Upsamples a surface by two and scales with given value. Uses a $2D$ thread array with a local work size of 32×4. All reads and writes are performed in terms of `float4`.

IV. RESULTS

The test setup consists of:

- **Host**: Intel Haswell Core i7 4770 CPU running at 3.9 GHz with 32 GB DDR3 DRAM
- **Device-1**: NVIDIA GTX-780 GPU running at 900 MHz with 3 GB GDDR5 DRAM (288.4 GB/s theoretical peak)
- **Device-2**: NVIDIA GT-640 GPU running at 900 MHz with 2 GB DDR3 DRAM (28.5 GB/s theoretical peak)
- **OpenCL**: NVIDIA OpenCL 1.2 implementation.

Reference MATLAB code was run on the Host CPU and the system was dedicated to testing with no other substantial work executing during test runs. OpenCL kernels were run on GTX-780 and GT-640 which were the only GPU type OpenCL devices present on the system (apart from the integrated Intel HD 4600 GPU that comes with Core i7 4770 CPU). We implemented the algorithm using both single (32 bit) and double (64 bit) precision floating point numbers.

TABLE I: Per frame execution times with GT-640 with single (32bit) precision

Resolution	CPU	GPU	Speed up
576×384	N/A	951.3 ms	N/A
640×480	N/A	1268.9 ms	N/A
1280×960	N/A	3809.4 ms	N/A

TABLE II: Per frame execution times with GT-640 with double (64bit) precision

Resolution	CPU	GPU	Speed up
576×384	107 sec	3889.3 ms	27×
640×480	156 sec	5224.8 ms	30×
1280×960	471 sec	15424.3 ms	31×

Since MATLAB is inherently double precision, we do not report any single precision timing results for CPU/MATLAB code. The benchmark results (worst -longest- execution times in milliseconds over 1000 separate runs) obtained for the RubberWhale sample from the Middlebury data set [7] are given in Tables I, II, III and IV. Data transfers (from CPU to GPU and vice versa) are included in the measured times. For every pyramid level 10 warps are executed with 50 iterations per warp. Constants are set as $\lambda = 40$, $\beta = 0.01$, $\tau = 1/\sqrt{8}$ and $\epsilon = 0$ (all in agreement with the distributed MATLAB source that is available at [11]).

The visual optical flow results including the ground truth, result of the original MATLAB code, double precision GPU result, and single precision GPU result can be seen in Figures 6, 7, 8 and 9, respectively. Color coding used in these visuals is based on representing the flow vector direction with hue and vector magnitude with saturation. The input images and the ground truth are downloaded from [3], where our specific test input image is labeled as RubberWhale. As it can be seen from the figures, single and double precision implementations compute very similar results and match the single threaded CPU implementation in terms of flow vector precision. For the RubberWhale test sample, the average endpoint errors [8] were measured as 0.143 for both double and single precision implementations. We could not observe any endpoint error differences between single and double precision implementations for other test samples (Dimetrodon, Grove2, Grove3, Hdrangea, Urban2, Urban3) as well. In terms of per frame execution time speed-up factors, GTX-780 achieved 298× and GT-640 achieved 31× for 1280×960 input frames. Based on our tests, flow vectors computed by the single precision implementation are comparable to the double precision implementation. Furthermore, the improvement achieved by the double precision implementation comes with increased execution times.

The performance comparison of GTX-780 and GT-640 is summarized in Tables V and VI, which present the per frame execution times and the resulting performance scaling factors. For the single precision implementation the performance scaling is around the expected factor of 6. For the double precision implementation, its around 9. The increasing difference in performance is mainly due to 10× larger DRAM bandwidth provided by GTX-780 (288 GB/s compared to 28.5 GB/s).

TABLE III: Per frame execution times with GTX-780 with single (32bit) precision

Resolution	CPU	GPU	Speed up
576×384	N/A	188.7 ms	N/A
640×480	N/A	230.6 ms	N/A
1280×960	N/A	528.9 ms	N/A

TABLE IV: Per frame execution times with GTX-780 with double (64bit) precision

Resolution	CPU	GPU	Speed up
576×384	107 sec	459.4 ms	232×
640×480	156 sec	604.7 ms	257×
1280×960	471 sec	1579.1 ms	298×

TABLE V: Per frame execution times with GT-640 and GTX-780 with single (32bit) precision

Resolution	GTX-780	GT-640	Factor
576×384	188.7 ms	951.3 ms	5.0×
640×480	230.6 ms	1268.9 ms	5.5×
1280×960	528.9 ms	3809.4 ms	7.2×

TABLE VI: Per frame execution times with GT-640 and GTX-780 with double (64bit) precision

Resolution	GTX-780	GT-640	Factor
576×384	459.4 ms	3889.3 ms	8.5×
640×480	604.7 ms	5224.8 ms	8.6×
1280×960	1579.1 ms	15424.3 ms	9.7×

V. Conclusion

We presented a massively multi-threaded GPU implementation of the anisotropic Huber-L^1 optical flow estimation algorithm proposed in [6] using OpenCL framework. Overall algorithm flow and GPU specific implementation details were discussed and performance results were presented.

Based on our analysis and the performance tests of the resulting OpenCL implementation, optical flow estimation algorithm originally proposed in [6] was found to be a good match for massively multi-threaded SIMD implementation. Depending on the input image size, per frame execution time speed-ups of 200-300× were observed. Single precision implementation provided results that are comparable to double precision implementation at lower per frame execution times.

Finally, we did not conduct any motion vector quality studies to detect minimal warp and iteration numbers, which have a direct impact on the overall per frame execution time. As a result, we simply executed with excessive warp (10) and iteration (50) numbers resulting in high execution times. Depending on the motion vector quality required by a specific processing block that will use the output of the optical flow estimation block, the warp and iteration numbers can be lowered resulting in much reduced per frame execution times.

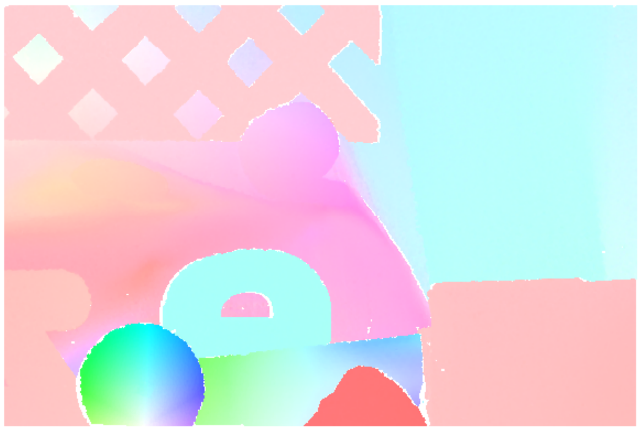
Fig. 6: Ground truth

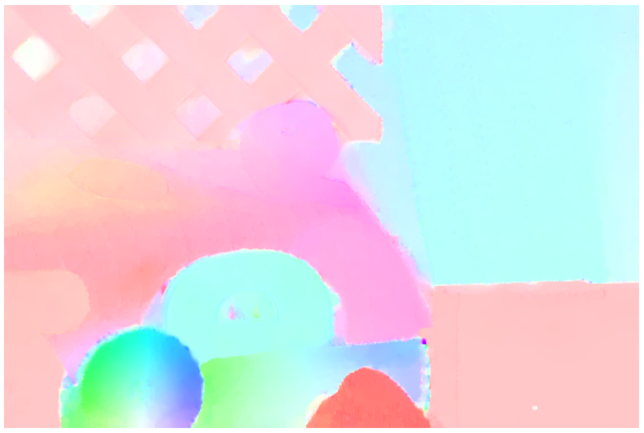
Fig. 8: GPU output - double precision

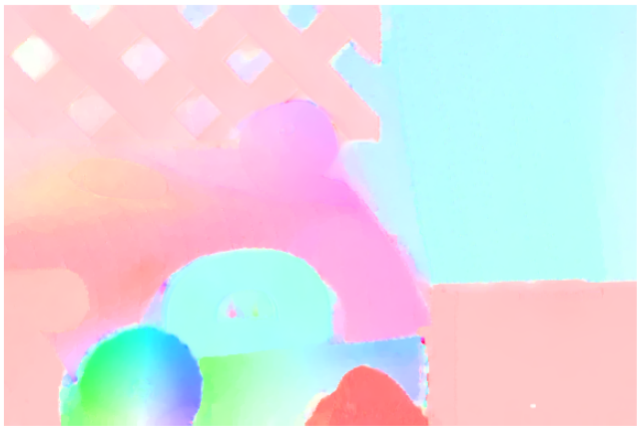
Fig. 7: CPU output

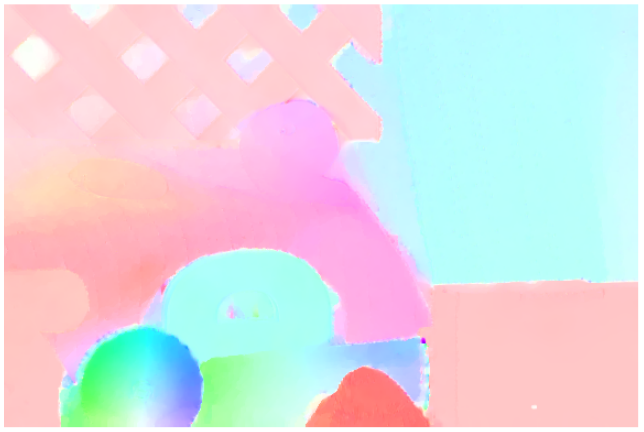
Fig. 9: GPU output - single precision

ACKNOWLEDGMENT

This work has been partially supported by the Artemis JU Project ALMARVI (Algorithms, Design Methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing), Artemis GA 621439 [13] and TÜBİTAK (The Scientific and Technological Research Council of Turkey) TEYDEB 1509 Program Project number 9140015.

REFERENCES

[1] B. D. Lucas and T. Kanade, *An iterative image registration technique with an application to stereo vision*, In Proceedings of the 7th International Joint Conference on Artificial Intelligence, pages 674679, April 1981.

[2] B. K. P. Horn and B. G. Schunck, *Determining optical flow*, Artificial Intelligence, v17, 185203, 1981.

[3] S. Baker, D. Scharstein, J.P. Lewis, S. Roth, M. J. Black and R. Szeliski, *A Database and Evaluation Methology for Optical Flow*, in International Journal of Computer Vision, 92(1):1-31, March 2011.

[4] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry and D. Schaa, *Heterogeneous Computing with OpenCL*, Morgan Kaufmann, 2011.

[5] A. Dore and S. Lasrado, *Performance Analysis of Sobel Filter on Heterogeneous System Using OpenCL*, NCRIET, Karnataka, India, May 2014.

[6] M. Werlberger, W. Trobin, T. Pock, A. Wedel, D. Cremers and H. Bischof, *Anisotropic Huber-L1 Optical Flow*, BMVC, London, UK, Sept.2009.

[7] Middlebury Optical Flow webpage, *http://vision.middlebury.edu/flow/*. Last accessed on May 2015.

[8] S. Baker, D. Scharstein, J.P. Lewis, S. Roth, M.J. Black and R. Szeliski, *A Database and Evaluation Methodology for Optical Flow*, International Journal of Computational Vision, 92:131, 2011.

[9] NVIDIA GeForce GTX-780 Specifications, *www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications*. last accessed on May 2015.

[10] NVIDIA, *http://www.nvidia.com.tr/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf*. Last accessed on May 2015.

[11] Graz University of Technology GPU4Vision, *http://www.gpu4vision.org*. Last accessed on May 2015.

[12] J. Rosner, H. Fassold, P. Schallauer and W. Bailer, *Fast GPU-based image warping and inpainting for frame interpolation*, International Conferences on Computer Graphics,Vision and Mathematics, Brno, Czech Republic, 2010.

[13] Artemis JU Project ALMARVI Algorithms, Design Methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing, GA 621439. http://www.almarvi.eu