Real-time Depth Estimation and View Interpolation using Quasar

Bart Goossens, Simon Donné, Jan Aelterman, Jonas De Vylder, Dirk Van Haerenborgh and Wilfried Philips (Ghent University - TELIN/IPI/iMinds)

1 Abstract

In this paper, we first present a new programming framework, Quasar, for high-level programming on heterogeneous CPU and single/multi-GPU systems. Quasar consists of a high-level language, a corresponding integrated development environment (IDE), a compiler, and a run-time system. Its aim is to relieve the programmer from hardware-related implementation issues that commonly occur in CPU/GPU programming. Examples include selecting memory types, memory management, concurrent kernel/function execution, hardware texturing units, etc. In doing so Quasar allows the programmer to focus on designing, testing and improving the image/video processing algorithms themselves, rather than their implementation. A practical application of this programming framework is presented: a new real-time multi-camera depth estimation algorithm and subsequent view interpolation. Currently, for view interpolation, we obtain a processing speed of 30 frames/sec for HD video based on two views with Quasar on an NVIDIA Geforce GTX 770 GPU.

2 Introduction

Graphical processing units (GPUs) are increasingly being used to complement CPUs for computationally intensive tasks with large amounts of data, such as typically encountered in image and video processing. The excellent performance of GPUs for parallel processing operations often yields speed-up factors of 10x-50x for image and video operations compared to a single-threaded CPU execution. Recently, there is also a trend towards the use of GPUs in embedded devices such as the NVIDIA Tegra.

Combining GPU programming with different sensors and platforms/devices is very challenging, because it requires specialized programming expertise. Furthermore, the resulting programs are not well amenable to algorithmic changes, which often require rewriting a large part of the code. For many developers, a typical programming workflow therefore consists of first implementing and testing the algorithm in a rapid-prototyping language (such as Octave/Matlab) and only later, when the algorithm is finalized, porting the algorithm to a native environment such as C++ with CUDA/OpenCL, which is generally time-consuming.

To improve the ease of programming on GPUs recently several efforts have been made. *Modular programming techniques* include existing software libraries such as Intel Array Building Blocks, NVIDIA Thrust, GPU-accelerated functions in OpenCV, Blitz++, Eigen, Armadillo, ...). Alternatively some *domain-specific languages* have been designed (e.g. Halide [1]). Other solutions include parallel extensions integrated in C/C++ OpenACC [2], Microsoft C++ `AMP` [3], and/or *programming languages* with integrated GPU support (e.g. Mozilla Rust [4]).

Despite these efforts, the corresponding development tools are not well suited for rapid prototyping. Particularly for researchers, GPU programming has several disadvantages: 1) there is a steep learning curve, 2) the implementation and optimization time can take several weeks to months even for a simple algorithm, 3) often different code paths must be written for different target platforms, or even different generations of the same platform (e.g. different GPU generations), 4) the testing and debugging of the code is not always trivial and 5) the programming code may not be future-proof: it is not guaranteed to work optimally on future CPU/GPU devices. A major cause of these disadvantages is that the algorithmic specification and its implementation are not separated: a programmer spends a lot of time on device-specific optimization of the implementation rather than algorithmic improvement. The Quasar programming framework is aimed at allowing abstraction of the algorithmic specification from implementation. It achieves this through automatic optimization both at compile-time and at run-time, thereby alleviating many of the aforementioned disadvantages.

The purpose of this paper is to demonstrate how a complex video-processing algorithm can be designed, implemented and tested with a heterogeneous computing architecture in mind, so that processing can be done in real-time. We start with an overview of the Quasar framework in Section 3. We then present and evaluate a Quasar implementation of a demanding image processing application to showcase the real-world advantage of using the Quasar framework in Section 4. Results and a discussion are given in Section 5. Finally, Section 6 concludes this paper.

The chosen application is *stereo depth estimation and view interpolation*. Here, stereo images are captured using two cameras arranged close to each other. The cameras are calibrated off-line, using a 2D pattern (e.g., a checkerboard). During on-line operation, left and right depth maps are estimated from the captured images. Next, intermediate views of non-existing camera views are calculated. With traditional methods (e.g., CUDA/OpenCL), the stereo-vision and view interpolation is particularly challenging to implement on a GPU due to the size and complexity of the algorithms (stereovision with hierarchical refinement, view synthesis, ...). A substantial effort from the programmer is required, especially when the algorithms need to be hybridly optimized for the specific CPU/GPU architecture. Some processing steps are best suited for the GPU, while other processing steps tend to be more efficient on the CPU. In our approach, program analysis and target-specific optimization is performed by the Quasar front-end compiler, and heterogeneous execution is obtained by dynamic run-time decisions, all performed transparently by the execution run-time; therefore the programmer can completely focus on the algorithms independently of the target device and the corresponding implementation.

3 A brief overview of the Quasar programming framework

The Quasar language is a high-level programming language with a syntax similar to Octave/MATLAB, making it easy to learn. The framework also contains a compiler and a run-time system. A schematic overview is given in Figure 1. The compiler consists of a front-end and several back-end compilers. The front-end compiler extracts certain code regions (e.g., loops via automatic parallelization, kernel functions, ...), and automatically generates target-dependent code that is then compiled using one of the back-end compilers. The front-end compiler also performs several high-level code optimizations, to aid the back-end code generation. Several existing commercial or open source back-end C/C++ compilers are currently supported: GCC, Clang, the Microsoft Visual C++ compiler, the Intel C++ compiler, the NVIDIA NVCC compiler (for CUDA) and the OpenCL compilers. Alternatively, an LLVM back-end allows to directly emit target-dependent binary code via the LLVM intermediate representation.

The generated binary code as well as the Quasar intermediate code is passed to the run-time system, which consists of four major components:

1. a **memory manager** that performs automatic memory management (allocation and deallocation, but notably the memory transfers between devices as well).
2. a **scheduler** that decides which device is used to perform a certain calculation (on a kernel function level).
3. a **load-balancer** will make sure each CPU/GPU thread is assigned is assigned sufficient work.
4. a **device manager** communicating with the underlying hardware through CUDA or OpenCL.

Typically, the compiler will generate code fragments for each distinct target device type (CPU/GPU) available, such that the run-time system is able to switch between devices as required for optimal execution speed. The run-time system makes these decisions by taking into account the current load of each device, the complexity of the task at hand and the cost to transfer the relevant memory blocks between devices. All of this is done in a fully automated fashion. The back-end compiler currently supports both CUDA and OpenCL devices, and may even be configured to use multiple devices simultaneously. This approach relieves the programmer from complex implementation issues (such as memory allocation and transfers, structure alignment and packing, use of texture memory, device selection, scheduling, etc), as this is handled transparently by the compiler and run-time system.

Additionally the Quasar runtime will select optimal execution parameters on the various devices. As an example, executing a kernel on the GPU typically requires the programmer to specify the block size, the number of threads, the amount of shared memory used by the kernel, and so on. The ideal values for these parameters not only differ from algorithm to algorithm, but also between devices. For GPU kernels this is dictated by the number of registers a kernel function requires, as well as the data dimensions. The Quasar run-time heuristically sets the parameters to good values [5].

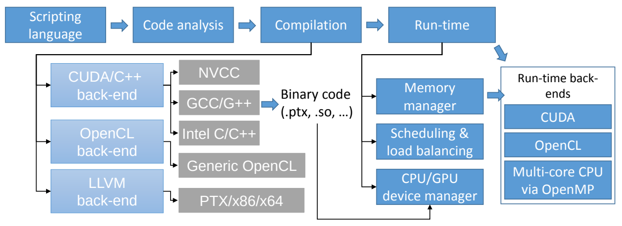

Figure 1. Overview of the Quasar architecture: the compiler and the run-time back-ends.

4 Real-time Depth Estimation and View Interpolation

As an application example, we design a new algorithm for real-time depth estimation and view interpolation from a stereo camera setup. The algorithm performs the following operations in sequence (see Fig. 2): Based on calibration matrices that are calculated off-line, imagery from both cameras is rectified in real-time. Next, a hierarchical disparity estimation method with optical flow-based total variation is used to estimate the depth images for the different views [6]. Finally, a view interpolation algorithm estimates intermediate views from the available RGB images and the previously estimated depth images. The rectification, depth estimation and view interpolation algorithms are all developed in Quasar and their implementation only requires a modest number of lines of code (based on experience with direct code porting between Quasar and C++/CUDA, about 3-10x less than equivalent C++ or CUDA code). This application is a well suited use-case for the design of more complex real-time video processing algorithms on heterogeneous computing architectures. In particular, we will show that for high number of scales, the hierarchical disparity estimation even requires hybrid processing on CPU and GPU in order to offer the best performance.

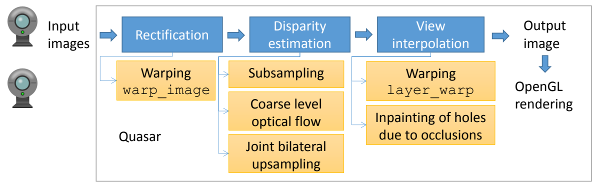

Figure 2. Overview of the depth estimation and view interpolation algorithm.

4.1 Implementation details

In this section, we describe the method and its implementation details in more detail.

Rectification The first required step is the calculation of rectifying homographies. These are transformations that project the left and right image onto a common image plane, in order to simplify stereo-vision. In order to perform depth estimation through stereo-matching, we need to find pixel correspondences between the left and the right view. To simplify the stereo-matching, we use the *epipolar* rectification, which ensures that pixel correspondences are on the same scan-line in both images. The rectification process is illustrated in Figure 3.

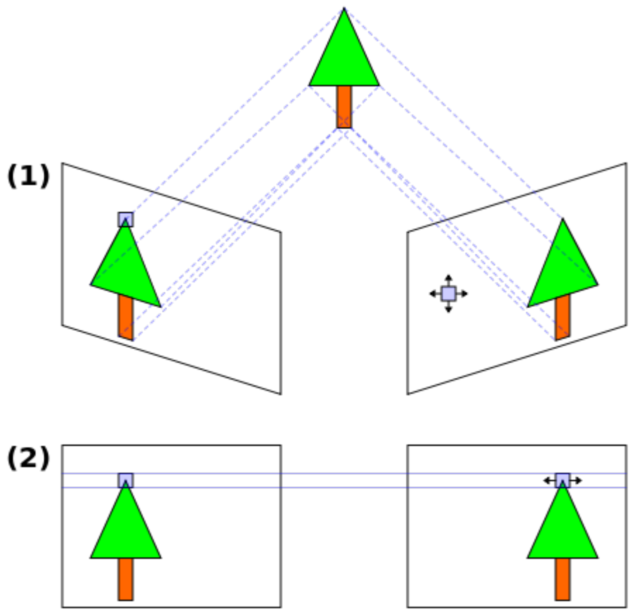

Figure 3. Illustration of stereo rectification: the images are warped with an affine transformation such that pixel correspondences are on the same scan-line in both images.

After computing the epipolar geometry, two affine transformations are obtained (i.e., one for each camera) which are used to warp the input views. For more details on the actual calibration, see for example Kumar et al. [7]. This warping is performed on a per-pixel basis, which is easily parallelized over the image domain: each pixel requires one matrix-vector multiplication ($(3 \times 4) \times (4 \times 1)$) and one indexing into the original image. The code fragment from Figure 4 illustrates the implementation. Here, the kernel function `invwarp_kernel` is applied in parallel to every pixel in the image, using the `parallel_do` function. In our example, we use warping with linear interpolation. Quasar allows to use GPU hardware textures for images with a width that is a multiple of 32, by the simple access modifier `hwtex_linear(4)`. Here, the parameter value 4 indicates that four color components are being used for the texture addressing (from which in this case only 3 components, R,G and B are used).

Hierarchical disparity estimation Using the rectified views, we estimate disparity (which is the inverse of scene depth) hierarchically, using a two-step approach at each hierarchical level: input views are downsampled to a lower resolution where disparity is first estimated, after which it is upscaled to higher resolutions in order to refine the estimate.

In the first phase, input views are down-sampled linearly (in the implementation a factor 4 is used). A coarse estimate of disparity is obtained from these images. This estimation involves an adapted version of the optical flow estimation technique from Drulea and Nedevschi [8]. Our adaptation is that we restrict the technique, which searches for correspondences in the entire image plane, to only allow matches on the same scanline. This low-resolution disparity estimation algorithm works on a per-pixel basis, which allows for much parallelism in each iteration of the optimization problem.

```
function [] = warp_image(input:cube'hwtex_linear(4),
    H:mat,bounding_box:mat,output:cube)
  [Mo,No] = size(output,0..1)

  function [] = __kernel__ invwarp_kernel(pos:ivec2)
    y_new = pos[0] + bounding_box[0,0]
    x_new = pos[1] + bounding_box[0,1]
    y_old = H[0,0]*y_new + H[0,1]*x_new + H[0,2]
    x_old = H[1,0]*y_new + H[1,1]*x_new + H[1,2]
    z_old = H[2,0]*y_new + H[2,1]*x_new + H[2,2]
    y_old /= z_old
    x_old /= z_old
    output[pos[0],pos[1],0..2] =
      input[y_old,x_old,0..2]
  end
  parallel_do([Mo,No],invwarp_kernel)
end
```

Figure 4. Example Quasar code - image rectification.

The second phase completes the coarse-to-fine approach: the coarse disparity estimates are upsampled to a higher resolution and refined. The upsample technique uses the dual-cross bilateral grid method based from Chen et al. [9]. It is a guided upsampling technique, which can be described as an approximated joint bilateral upsampling that is computationally more efficient. These cross-bilateral methods use a so-called *guide image*, to regularize the upsampling of a target image. In our case, the color input left and right views are used as guide images for the disparity estimates. In this way, only real edges, i.e., image edges that are present in the color input images, are preserved in the disparity upsampling, resulting in a realistic and desirable upsampling result. This upsampled result is then refined at the higher resolution.

Depending on the hierarchical depth of this process, it may occur that the dimensions of the coarsest (i.e., most downsampled) images are too small to yield efficient execution on a GPU, due to the fact that the resulting occupancy of the GPU may become too low. This implementation aspect is automatically handled by the Quasar run-time system, which will detect this case and perform the calculations for the coarsest scale on the CPU (possibly using OpenMP for exploiting the parallelism). This way, the overall algorithm efficiently exploits the heterogeneous nature of the CPU/GPU combination.

View interpolation Using disparity estimates for both images, views positioned on the line segment between both cameras are reconstructed. As we obtained the disparity fields u_L and u_R, we can state that these (ideally) relate each left image $I_L(x,y)$ with each right image $I_R(x,y)$ via:

$$\begin{aligned} I_L(x,y) &= I_R(x - u_L(x,y), y), \quad \text{and} \\ I_R(x,y) &= I_L(x + u_R(x,y), y). \end{aligned} \tag{1}$$

We wish to reconstruct an intermediate view I_θ at location θ (defining the left image as $I_0 = I_L$ and the right image as $I_1 = I_R$). Thanks to epipolar geometry, we have the following relationships:

$$\begin{aligned} I_L(x,y) &= I_\theta(x - \theta u_L(x,y), y), \quad \text{and} \\ I_R(x,y) &= I_\theta(x + (1-\theta) u_R(x,y), y). \end{aligned} \tag{2}$$

These relationships are used to warp both input images, using the estimated per-pixel disparity the required location θ of the interpolated view. The warping needs to be performed in such a way that each pixel in the interpolated view is filled in by at least one pixel value from the input views. Three situations can occur: 1) there is exactly one input pixel warping to the pixel location in the interpolated view, 2) there is a collision (i.e. multiple input pixels warping to the same pixel location in the interpolated view) or 3) there is a hole (i.e. no input pixels warp to a given pixel location in the interpolated view).

To avoid collisions, the implementation of the warping uses a z-buffer (built using atomic max-operations) to ensure that the write collisions in the reconstructed view image are resolved in favor of the closest object.

Such situations are typically caused by (de)occlusions, which are also the main reason for holes. Such holes, which tend to be small, are inpainted using a simple directional search: each missing pixel looks into the 8 cardinal directions until it finds a pixel that has valid data, and weighs these 8 ”neighbors” according to their distance from its location. To a certain extent this is even possible for reconstructing view positions past either end point of the line segment (view extrapolation), but this requires the occasional inpainting of large areas and quickly breaks down as the distance from the line segment increases. Note that it is also possible to warp the disparity estimates of both views onto the sought-after location, resulting in a disparity estimation of the reconstructed view.

The warping code is illustrated in Fig. 5. The code uses two 2D parallel for loops. In the first loop, the z-buffers for the left and right frames are calculated. In the second loop, the images are warped according to the z-buffers. During compilation, the Quasar compiler converts the parallel loop to a kernel function, that is then further compiled using a back-end compiler (such as the NVIDIA NVCC compiler).

The hole inpainting code (which is too long to include in this paper), works on similar principles. The code performs another 7 passes over the image, while inpainting the disparity. Disparity values that need to be inpainted are nearly always the result of de-occlusions: they are hence filled in with a weighted sum of the lower half of the disparities in their neighbourhood.

5 Results and discussion

To demonstrate the effectiveness of Quasar in the efficient and time-effective implementation of algorithms, we will now compare timing results of different Quasar implementations. The view synthesis algorithm's Quasar implementation is run on different hardware platforms, where the Quasar compiler and runtime system are responsible for automatic code optimization. The timing results as shown in Table 1 are averaged over 100 iterations of the algorithm. These timing results showcase the vast acceleration potentional that can be expected from the hardware capabilities of a GPU device. The benefit from Quasar comes from the fact that this potential is unlocked from a source code complexity that is similar to what a matlab or python implementation would be, without the need for manual hardware-specific code optimization. Achieving such speedup with dedicated CUDA code would have result in far higher code complexity and loss of generality due to the necessity of platform-specific optimization by the programmer.

Timing breakdown	Rect.	Disp.	View synth.
CPU (i7-3930K)	67ms	24s	1000ms
CPU (i7-3930K) - openMP	45ms	11s	620ms
Quasar (GTX 770)	2ms	1.7s	8ms
Quasar (GT 440)	3ms	6.5s	45ms

Here, GTX 770 and GT 440 corresponds to respectively a Kepler-architecture level NVIDIA Geforce GTX 770 GPU and a Fermi-architecture level NVIDIA Geforce GT 440.

Using Quasar requires no development efforts when switching to different platforms. Because of this, obtaining realistic comparisons of performance across different platforms becomes trivially easy. This ease of use was exploited to create Table 1.

Finally, visual results of the resulting method can be seen in Figure 6. Despite a few minor distortions (see for example the anaglyph glasses in the middle of the picture), the overall visual depth and view interpolation quality is good.

It is also worth noting that the Quasar run-time decides on-the-fly whether to execute any given function on the CPU or on the GPU. The trade-off is made between the GPU's superior processing speed for massively parallel execution and the overhead involved with kernel launching and possible memory transfers to the GPU and back.

6 Conclusion

In the domain of image/video/multi-camera processing, Quasar allows researchers to focus on design aspects of the algorithms rather than implementational details. Our approach enables 1) fast hybrid execution on CPU/GPU, 2) fast rapid-prototyping with simplified debugging in a specialized IDE and 3) a future-proof methodology (multiple GPU technologies are supported). In particular, we demonstrated the capabilities through a real-time depth estimation and view interpolation application. More information about Quasar is available at `http://quasar.ugent.be`.

7 Acknowledgments

Part of the research leading to this work was performed within the iMinds Hi-Tech Visualization (HiViz) strategic research program. B. Goossens acknowledges support through a Postdoctoral scholarship of FWO Flanders.

References

[1] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines,” in *ACM Trans. Graphics*, vol. 31, no. 4, 2012.

[2] *OpenACC - Directives for Accelerators*, Std., online: http://www.openacc.org.

[3] Microsoft, “C++ AMP,” online: http://msdn.microsoft.com/en-us/library/hh265137.aspx.

[4] “Mozilla Rust,” online: http://www.rust-lang.org.

[5] B. Goossens, “Program execution on heterogeneous platform,” WO Patent 2 015 150 342, 2015.

[6] S. Donné, B. Goossens, J. Aelterman, and W. Philips, “Variational multi-image stereo matching,” in *IEEE International Conference on Image Processing (ICIP2015)*, Québec City, Canada, Sept 27-30 2015.

```
function [] = layer_warp(image_left:cube, disparity_left:mat, image_right:cube,
    disparity_right:mat, location:scalar)
    [M,N,C] = size(image_left,0..2)
    [zbuffer_high_left,zbuffer_low_left,zbuffer_high_right,zbuffer_low_right] =
        {zeros(M,N)-1,zeros(M,N)+1e9,zeros(M,N)-1,zeros(M,N)+1e9}
    [warped_upper_left,warped_lower_left,warped_upper_right,warped_lower_right]  =
        {zeros(M,N,C),zeros(M,N,C),zeros(M,N,C),zeros(M,N,C)}

    {!parallel for}
    for y=0..M-1 % Generate z-buffers
        for x=0..N-1
            if disparity_left[y,x] >= 0  %from left to virtual
                x_virt = int(x - location*disparity_left[y,x])
                atomic_max(zbuffer_high_left[y,x_virt], disparity_left[y,x])
                atomic_min(zbuffer_low_left [y,x_virt], disparity_left[y,x])
            endif
            if disparity_right[y,x] >= 0   %from right to virtual
                x_virt = int(x + (1-location)*disparity_right[y,x])
                atomic_max(zbuffer_high_right[y,x_virt], disparity_right[y,x])
                atomic_min(zbuffer_low_right [y,x_virt], disparity_right[y,x])
            endif
        end
    end
    zbuffer_low_left[zbuffer_low_left == 1e9] = -1
    zbuffer_low_right[zbuffer_low_right == 1e9] = -1

    {!parallel for}
    for y=0..M-1
        for x=0..N-1
            %from left to virtual
            x_virt = int(x - location*disparity_left[y,x])
            if zbuffer_high_left[y,x_virt] == disparity_left[y,x]
                warped_upper_left[y,x_virt,0..2] = image_left[y,x,0..2]
            elseif zbuffer_low_left[y,x_virt] == disparity_left[y,x]
                warped_lower_left[y,x_virt,0..2] = image_left[y,x,0..2]
            endif
            %from right to virtual
            x_virt = int(x + (1-location)*disparity_right[y,x])
            if zbuffer_high_right[y,x_virt] == disparity_right[y,x]
                warped_upper_right[y,x_virt,0..2] = image_right[y,x,0..2]
            elseif zbuffer_low_right[y,x_virt] == disparity_right[y,x]
                warped_lower_right[y,x_virt,0..2] = image_right[y,x,0..2]
            endif
        end
    end
end
```

Figure 5. *Example Quasar code - calculation of z-buffers and image warping for view interpolation.*

[7] S. Kumar, C. Micheloni, C. Piciarelli, and G. Foresti, "Stereo rectification of uncalibrated and heterogeneous images," *Pattern Recognition Letters*, vol. 31, no. 11, pp. 1445 – 1452, 2010.

[8] M. Drulea and S. Nedevschi, "Motion estimation using the correlation transform," *IEEE Trans. Image Processing*, vol. 22, no. 8, pp. 3260–3270, 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6516084

[9] J. Chen, S. Paris, and F. Durand, "Real-time edge-aware image processing with the bilateral grid," *ACM Trans. Graph.*, vol. 26, no. 3, Jul. 2007. [Online]. Available: http://doi.acm.org/10.1145/1276377.1276506

Author Biography

Bart Goossens earned his master's degree in Computer Science Engineering from Ghent University, Belgium in 2006 and the Ph.D. degree from the same university in 2010. Since Oct. 2010 he is a postdoctoral fellow of FWO Flanders and since Oct. 2013 he is also a part-time lecturer (Professor) at Ghent University. For his research, he received the Barco/FWO prize for master theses 2006 and the Scientific prize IBM Belgium for Informatics 2011. Since 2011, he has been developing Quasar, a programming language and framework with IDE, to ease the de-

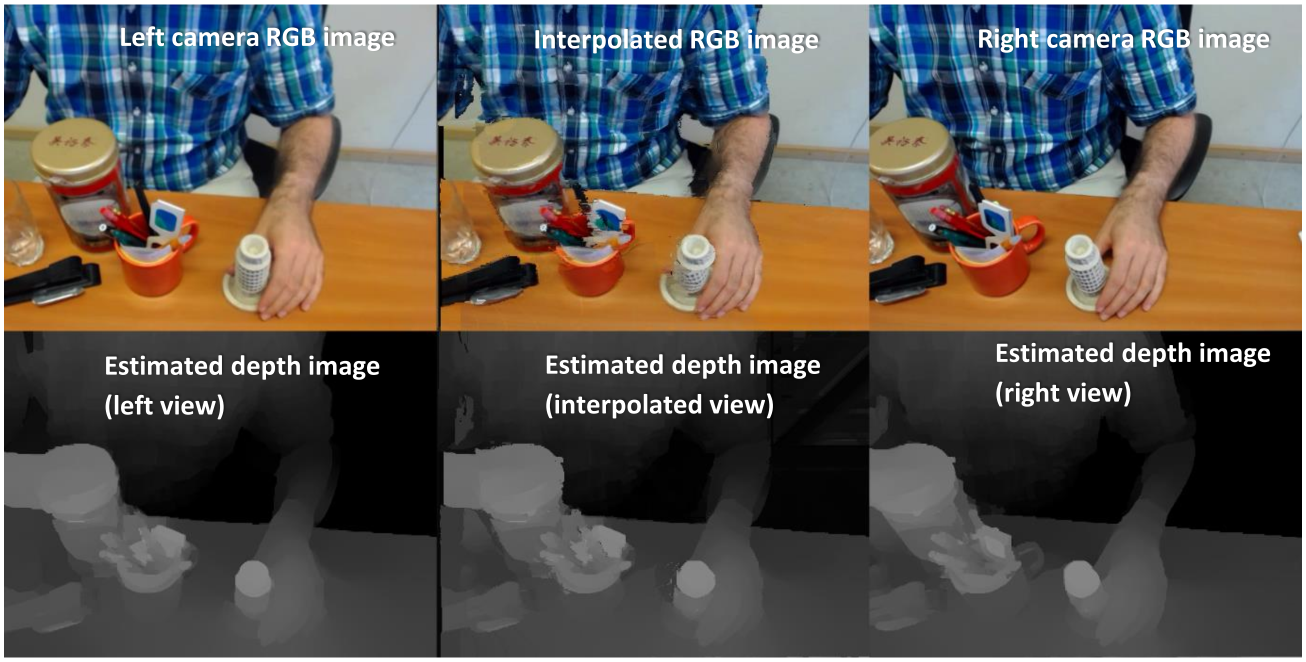
Left camera RGB image
Interpolated RGB image
Right camera RGB image
Estimated depth image (left view)
Estimated depth image (interpolated view)
Estimated depth image (right view)

Figure 6. *Visual results of the depth estimation and view interpolation*

velopment of complex image and video processing algorithms for heterogeneous devices (e.g., GPU and multi-core CPU).

Simon Donné earned a master's degree in Computer Science Engineering at Ghent University in 2013. He is currently employed as a researcher at IPI-iMinds research group funded by the Special Research Fund of Ghent university (grant number 01D21213) in preparation of his Ph.D. His research interests include multiview image processing and 3D reconstruction.

Jan Aelterman holds a master's degree in Electrical Engineering and a Ph.D. degree in engineering from Ghent University Belgium. As of 2015, he is working as a postdoctoral researcher at the IPI-iMinds research group at Ghent University. His research interests include image restoration, compressed sensing and multiview image processing.

Jonas De Vylder received his masters degree in formal informatics at Ghent University in 2007. After graduation he joined the IPI-iMinds research group at Ghent University funded by an IWT research scholarship. In 2014 he earned a Ph.D degree for work focusing on model-based image analysis for biological applications and microscopy. Since 2015 he is working with the Quasar team (a collaboration between UGent and iMinds), where he is responsible for load balance modelling of kernels on heterogeneous hardware.

Dirk Van Haerenborgh earned his master's degree in Industrial Engineering at Artesis University College Antwerp in 2009. He is currently employed as a research engineer at the IPI-iMinds research group of Ghent University. His research interests include multicamera and video processing. Since 2014, he is part of the development team of the Quasar framework.

Wilfried Philips is a full-time professor in Ghent University and is heading the research group "Image Processing and Interpretation", which is also part of the Flemish ICT research institute iMinds. He received the Diploma degree in electrical engineering from Ghent University, Belgium, in 1989. Since October 1989 he has been working at the Department of Electronics and Information Systems of Ghent University. In 1993 he obtained the Ph.D. degree in electrical engineering from Ghent university, where he now heads the Image processing and Interpretation research group. His main research interests are image and video restoration, image analysis, and lossless and lossy data compression of images and video and processing of multimedia data.