Demo: Quasar - a New Programming Framework for Real-Time Image/Video Processing on GPU and CPU

Bart Goossens, Jonas De Vylder, Simon Donné and Wilfried Philips
Ghent University - TELIN/IPI/iMinds
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
bart.goossens@telin.ugent.be

ABSTRACT

In this demonstration, we present a new programming framework, Quasar, for heterogeneous programming on CPU and single/multi-GPU. Our programming framework consists of a high-level language that is aimed at relieving the programmer from hardware-related implementation issues that commonly occur in CPU/GPU programming, allowing the programmer to focus on the specification, the design, testing and the improvement of the algorithms. We will demonstrate a real-time multi-camera processing application using our integrated development environment (IDE). The IDE offers various image/video processing-related debugging functions and performance profiling features.

Keywords

GPU programming, Real-time video processing.

1. INTRODUCTION

Recently, graphical processing units (GPUs) are increasingly being used to complement CPUs in computationally intensive calculation tasks with large amounts of data, such as in image and video processing. This is due to the excellent performance of GPUs for parallel processing, often yielding speed-up factors of 10x-50x for image and video operations, compared to a single-threaded CPU execution. Recently, there is also a trend toward the use of GPUs in embedded devices, such as the NVidia Tegra system. Combining GPU programming with different sensors and platforms/devices is very challenging, requiring specialized programming expertise. Furthermore, the resulting programs are not well amenable to algorithmic changes, which often require rewriting a large part of the code. In practice, a typical programming workflow therefore consists in first implementing and testing the algorithm in a rapid-prototyping language (such as Octave/Matlab) and later porting the algorithm to a native environment such as C++ with `CUDA` [1]/OpenCL [2], which is generally time-consuming.

For a researcher, 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 GPUs, 4) the testing and debugging of the code is not always trivial and 5) the programming code is not future-proof in general: it is not guaranteed to work optimally on future CPU/GPU devices. A major issue is that the algorithmic specification and its implementation are *not separated:* a programmer spends a lot of time on implementation details rather than improving the algorithm.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICDSC '15 September 08-11, 2015, Seville, Spain
© 2015 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-3681-9/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2789116.2802654

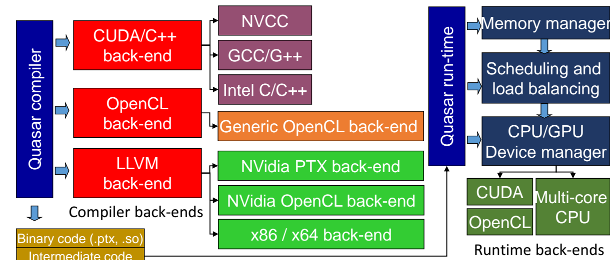

Figure 1: Overview of the Quasar architecture: compiler and run-time back-ends.

2. RELATED WORK

To improve the programmability of multi-core CPUs and GPUs recently several efforts have been made. These include 1) *modular programming techniques* (existing software libraries such as Intel Array Building Blocks, NVidia Thrust, GPU-accelerated functions in OpenCV, Blitz++, Eigen, Armadillo, ...), 2) *domain-specific languages* or parallel extensions integrated in C/C++ (e.g., Halide [3], OpenACC [4], Microsoft C++ `AMP` [5], ...) and 4) *programming languages* with integrated GPU support (e.g., Mozilla Rust [6]). In general, many of these approaches require a substantial effort from the programmer and often fixed programming patterns are enforced. Moreover, it is difficult to share programming code between different researchers.

3. OVERVIEW OF QUASAR

Quasar is a high-level programming language with a syntax that similar to Octave/MATLAB (see Figure 3), so that it is easy to learn. The framework also contains a compiler and a run-time system. The compiler extracts certain code regions (e.g., loops via automatic parallelization, kernel functions, ...) and it automatically generates target-dependent code that is then compiled using one of the back-

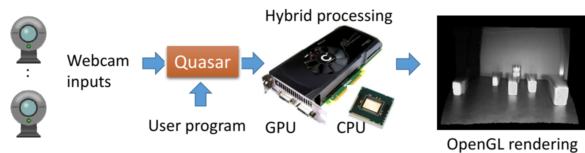

Figure 2: Demo setup: input/output chain of the Quasar program.

end compilers. An overview of the Quasar architecture is shown in Figure 1. A back-end compiler can be an existing C/C++ compiler (such as GCC, MSVC, Clang, ICC, ...), the NVidia compiler (for CUDA) or an OpenCL compiler. Alternatively, there is an LLVM [7] back-end that allows to directly emit target-dependent binary code via the LLVM intermediate representation. The generated binary code and also the Quasar intermediate code is passed to the run-time system, which consists of a **memory manager** (performing automatic memory management), a **scheduler and load-balancer** (which makes dynamic run-time decisions on what device to use for which task), and a **device manager** (which communicates with the underlying hardware through CUDA or OpenCL). Each extract code fragment is for example compiled for each target device. At run-time, the load-balancer then decides which version (e.g., CPU or GPU) of the code to run, depending on the current state.

4. DESCRIPTION OF THE DEMO

In this demo, we will illustrate the advantages of using Quasar for designing camera-processing algorithms. We will focus on real-time processing of multi-camera images (see Figure 2), in which a 3D version of the scene is constructed based on a program written in Quasar. The 3D representation is rendered via OpenGL (through CUDA/OpenGL interoperability that is integrated in Quasar). Spectators will be able to watch the 3D reconstruction algorithm in real-time. Next, also different aspects of the IDE will be demonstrated (the code editor, various debugging windows and tooltips and performance profiler with a timeline view).

Example Quasar programming code is given in Figure 3. The illustrated code calculates a z-buffer based on a disparity map (used for view interpolation). Noteworthy is the `parallel_do` function, which is used for launching the kernel function `generate_zbuffers` in parallel (either on CPU or GPU, a dynamic load-balancing decision done by the run-time). The code uses atomic minimum and maximum operators, which are hardware-accelerated.

Within the IDE (see Figure 4), the user can choose the device to run the code (e.g., the CPU/GPU) and various run-time settings. User programs can be started and paused, allowing the user to step through the code, place breakpoints, visualize intermediate images and values. There is also the possibility of parallel debugging through a software-based GPU emulation.

5. CONCLUSION

In the domain of image/video/multi-camera processing, Quasar allows researchers to focus on design aspects of the algorithms rather than implementation aspects. 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). Our methodology has successfully been used within several projects of the IPI research group, including 4 iMinds ICON projects, an IWT SBO project and 2 European projects.

```
function [zbuffer_high, zbuffer_low] = calc_zbuffer(
    disparity:mat, location:scalar)
[M,N] = size(disparity,0..1)
zbuffer_high:mat  = zeros(M,N)
zbuffer_low:mat   = zeros(M,N)+1e9

function [] = __kernel__ generate_zbuffers(pos:ivec2)
 [y,x] = pos
 if disparity[y,x] > 0 % disparity image
  x_virt = int(x - location*disparity[y,x])
  if x_virt >= 0 && x_virt < N
   atomic_max(zbuffer_high[y,x_virt],disparity[y,x])
   atomic_min(zbuffer_low [y,x_virt],disparity[y,x])
  endif
 endif
end
parallel_do([M,N],generate_zbuffers)
end
```

Figure 3: Example Quasar code - calculation of low and high z-buffers for view interpolation.

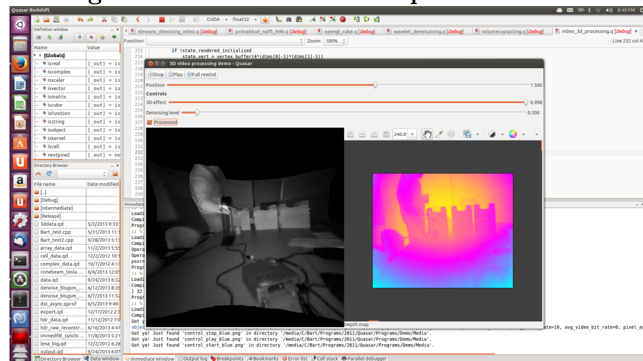

Figure 4: Screenshot of Quasar Redshift IDE - demonstrating a real-time 3D rendering of a video sequence.

Currently, the framework is in a prototype phase at UGent/iMinds, in which about 35 Ph.D. students and postdocs (from 4 universities) are testing the tools in different image processing application domains (e.g. 3D reconstruction, registration, computer vision, medical image reconstruction). More information is available at http://quasar.ugent.be.

6. ACKNOWLEDGMENTS

Bart Goossens acknowledges support by a postdoctoral fellowship of the Research Foundation–Flanders (FWO, Belgium).

7. REFERENCES

[1] NVidia, "NVidia CUDA Compute Unified Device Architecture," 2007, online: http://www.nvidia.com.
[2] *The OpenCL Specification 1.2*, Khronos OpenCL working group Std., 2011, online: http://www.khronos.org.
[3] 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.
[4] *OpenACC - Directives for Accelerators*, Std., online: http://www.openacc.org.
[5] Microsoft, "C++ AMP," online: http://msdn.microsoft.com/en-us/library/hh265137.aspx.
[6] "Mozilla Rust," online: http://www.rust-lang.org.
[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004, pp. 75–88, online: http://llvm.org.