QUASAR - A NEW HETEROGENEOUS PROGRAMMING FRAMEWORK FOR IMAGE AND VIDEO PROCESSING ALGORITHMS ON CPU AND GPU

Bart Goossens, Jonas De Vylder and Wilfried Philips

Ghent University - TELIN - IPI - iMinds
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
bart.goossens@telin.ugent.be

ABSTRACT

In image and video processing research, rapid prototyping and testing of different variations of an algorithm is quite essential (e.g., to find out if a given algorithm can solve a given problem or work in real-time). In the past decade, the computational performance of graphical processing units (GPUs) has improved significantly, where speed-up factors of 10x-50x compared to single-threaded CPU execution are not uncommon. However, GPU programming is challenging, requiring a significant programming expertise and moreover, most existing programming approaches are not well suited for rapid prototyping.

In this Show & Tell session, we present a new programming framework, aimed at making the bridge between high-level program specification and low-level implementation and optimization on heterogeneous computation devices. The goal is that the programmer is relieved from (most) implementation issues and can focus on the specification and improvement of the algorithms. We present a new prototype domain-specific programming language (in the first place aimed at image and video processing) that provides a uniform programming approach toward different hardware devices, a run-time environment to manage and communicate with the heterogeneous devices and an integrated development environment (IDE). The IDE provides various useful image processing-related debugging and profiling features.

I. INTRODUCTION

In image processing, rapid prototyping is very important: it is crucial to quickly verify different approaches for a given problem, to determine whether certain techniques can work for a specific problem and/or an a specific computing platform. Recently, there has been a lot of interest in multi-core Central Processing Unit (CPU), Graphical Processing Unit (GPU) and many-core coprocessor (accelerator) programming. In particular, GPUs are very well suited for algorithms that exhibit massive parallelism and that have a non-diverging control flow (such as many image processing algorithms). This often leads to significant speed-ups compared to single-core CPU implementations. The speed-up of the computations not only means that certain algorithms can work in real-time, but also that various modifications (e.g., different parameter settings) can easily be tested. Unfortunately, GPU/accelerator programming comes at a cost: 1) the sophisticated programming and debugging techniques lead to a steep learning curve for programmers that are new to these platforms, 2) development and optimization often requires huge efforts from the programmer (e.g., several weeks to months for implementing a relatively "simple" algorithm on a GPU), 3) often different versions of the code for different target platforms need to be written, 4) the resulting code may not be future-proof: it is not guaranteed to work optimally on future devices. The essential cause of the problem is that the algorithmic specification and its implementation are not *separated*: a programmer spends a huge amount of time focusing on implementation details rather than on improving the algorithm itself.

Consequently, the computational power of these platforms is often restricted to more experienced programmers and is not in reach of, e.g., starting Ph.D. students (at least, not without considerable investments).

Nevertheless, in the past decade, there has been a lot of work on designing new libraries and programming languages to simplify the programming tasks. A few options are:

1) The use of a low-level programming language (e.g., `CUDA` [1], OpenCL [2], DirectCompute [3]) in combination with a "host" programming language (e.g., C++, `FORTRAN`, Java, Ruby, Python, Octave, `MATLAB`, ...).
2) Modular programming techniques: based on building blocks in existing libraries (e.g., Array Building blocks, Thrust, `CUSP`, `MAGMA`, `FLAME`, GPU-accelerated functions in OpenCV, Armadillo, Blitz++, Eigen, ...)
3) *Domain-specific languages* and/or parallel extensions directly embedded into the high-level programming language (e.g., Halide [4], KernelGen [5], Rootbeer [6], OpenACC [7], Microsoft C++ `AMP` [8], ...). The DSLs and parallel extensions have the advantage of being easy to learn, while enabling easy integration in existing code bases.
4) Programming languages with integrated support (e.g., Mozilla Rust [9]).

Despite of the efforts, most of these techniques require a fair amount of programming expertise, a lot of manual work and often enforce fixed programming patterns (which is less flexible). Moreover, it is often difficult to share code between different researchers.

In this Show & Tell session, we present and demonstrate a new programming framework, called Quasar, that is (in the first place) aimed at image/video processing. The framework consists of a prototype of a programming language and an integrated development environment (IDE). The main goal is to hide implementation complexities so that the programmer can focus on the algorithmic design, while still giving (more experienced) programmers full control of the underlying implementation. The language itself is designed to be easy in use and has a syntax that is similar

(but not equal) to the syntax of MATLAB/Octave (see Figure 1). Moreover, the language provides a uniform programming approach toward CPUs and GPUs and is largely hardware-agnostic. Hence, programmers do not need to learn GPU concepts in order to take advantage of the acceleration.

Thanks to type inference, data type specifications can be omitted and can be inferred from the context. The language supports both dynamic and static typing. The development environment provides an interactive execution, integrated CPU/GPU debugger, advanced code profiling tools and documentation tools.

In summary, compared to existing frameworks, our approach provides both 1) fast hybrid execution on CPU/GPU, 2) fast rapid-prototyping with simplified debugging and 3) a future-proof methodology (multiple GPU technologies are supported). The *innovative* aspect is that image and video processing algorithms can be designed and developed in novel ways (e.g., with a lot of user interaction and visualization, tooltips specific to image/video processing, video probing debugging functions, ...). During the demonstrations, we will show these aspects in great detail. We aim for a lot of interaction with the audience: we will discuss the limiting factors in the design and development efforts of image/video processing algorithms, the approaches that they use to solve these problems, our solution. Finally, we hope to obtain feedback from the audience in order to further improve and extend our research platform.

In the following sections, we will discuss the different elements of our solution in more detail.

II. THE QUASAR PROGRAMMING FRAMEWORK

The Quasar framework consists of a compiler system, a runtime library and an IDE. The compiler system itself contains a front-end compiler and several back-end compilers. The front-end compiler translates high-level constructs in low(er)-level constructs that can be handled by the back-end compilers. The front-end compilation automatically detects and extracts serial and parallel loops that can be executed on the target device (e.g., CPU or GPU). In the front-end, also device-independent "host code" is generated to invoke the extracted parallel loops on the target device. The obtained intermediate Quasar representation is then either compiled to byte-code, or interpreted directly. The back-end compilers generate C++/CUDA or LLVM code and invoke existing native compilers (e.g. GCC, Clang, ...), to generate a device-dependent binary.

The Quasar runtime system consists of four major components: 1) a **memory manager** (that performs automatic memory allocation/deallocation/transfer between devices), 2) a **scheduler** (for deciding on which device a certain loop is executed), 3) a **load-balancer** (for making sure that each GPU/CPU thread has sufficient work) and 4) a **device back-end** (which communicates with the underlying hardware through CUDA or OpenCL). Finally, the results can be visualized via OpenGL (possibly via user interaction) or written to disk.

By default, code is generated for both CPU and GPU, so that the run-time system can dynamically switch between CPU and GPU, depending on the current load (load balancer), the complexity of the task (e.g. a 1D loop that iterates over 5 elements versus a 3D loop that iterates over all pixels of a volumetric image) and the memory transfer costs. This is all done fully automatically. The

```
num_it = 1024
im = zeros(768,768)

% Automatically parallelized loop (dim=2)
for m=0..size(im,0)-1
    for n=0..size(im,1)-1
        p = ([m,n] ./ size(im,0..1))-0.5
        c = -1.42 + complex(p[1],p[0])
        z = 0i
        N = 2.0
        for u=1..num_it
            if abs(z) > N
                break
            endif
            z = z * z + c
        end
        im[m,n] = u-log2(log(abs(z))/log(N))
    end
end

imshow(im,[])
```

Fig. 1. Example Quasar code - calculation of the *Mandelbrot* fractal.

run-time system currently supports both CUDA and OpenCL back-ends and can even use multiple CUDA and/or CUDA devices at the same time. This way, the programmer is relieved from complicated implementation issues (e.g., memory allocation, memory transfer, structure alignment and packing, device selection, scheduling, ...). Instead, these issues are handled transparently by the compiler and runtime systems. Several special CUDA features are supported and do not require special intervention of the programmer: hardware texturing units, asynchronous streaming interface, shared memory and dynamic parallelism.

A simple example Quasar program that calculates a *Mandelbrot* fractal and that displays the result, is given in Figure 1. A few simple programming language features can be noted: 1) built-in support of complex number arithmetic, 2) vector/matrix operations (./), 3) sequences a..b. These features are automatically mapped by the compiler onto equivalent low-level constructs: the complex number arithmetic and fixed-length vectors (e.g. [m,n]) are translated in single instruction/multiple data (SIMD) instructions by the back-end compiler (e.g. GCC, LLVM, ...).

III. THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

To simplify the development of new image and video processing algorithm, an IDE (called Quasar Redshift) has been designed (see Figure 2). The IDE has a classical (i.e., Eclipse/Scilab-like) layout and features a definition window (for inspecting variables and definitions), a data window (data visualization), a code editor, an interactive window, various debugging windows and a profiler with a timeline view. Several facilities are available for image/video processing, including debug tool-tip windows with image visualization, GPU windows, image and video probes, ... The user can choose the target device(s) to run the code, the desired numerical precision (e.g. 32-bit or 64-bit floating point). Debugging functionality includes starting and pausing programs,

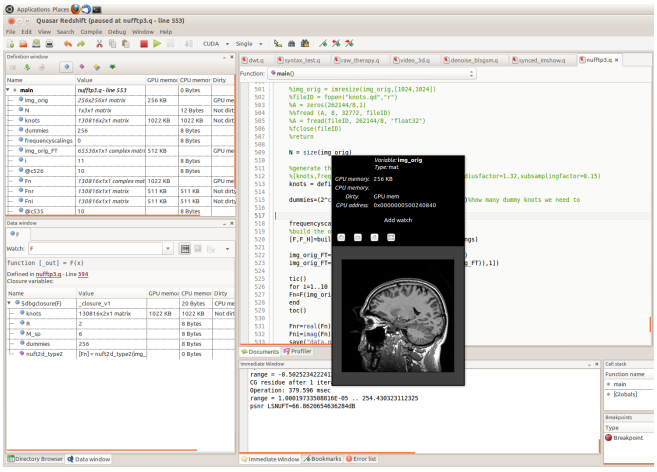

Fig. 2. Screenshot of Quasar Redshift (host OS: Ubuntu 12.4) - MRI reconstruction demo.

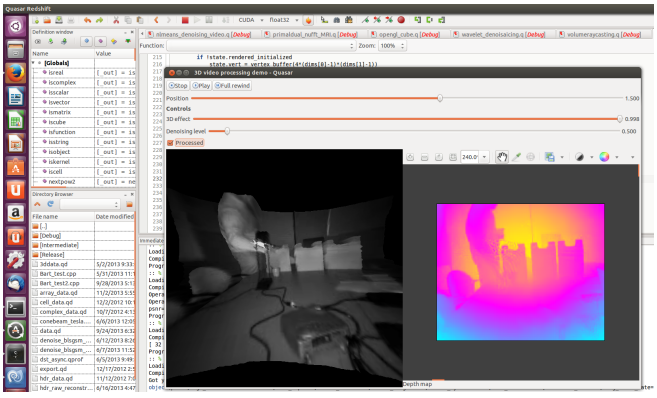

Fig. 3. Screenshot of Quasar Redshift (host OS: Ubuntu 13.4) - 3D video processing demo.

stepping through code, breakpoints, parallel debugging through GPU emulation. We will demonstrate these features through an interactive 2D+depth image reconstruction and visualization demo program. This program, completely written in Quasar, demonstrates real-time video processing and visualization through OpenGL triangle mesh rendering. Various sliders allow the user to interactively change parameters of the depth reconstruction algorithm and the results are seen immediately (see Figure 3).

IV. NOVEL RESEARCH OPPORTUNITIES

Our solution offers a number of novel research opportunities for the image and video processing community, that are difficult to accomplish with other techniques: 1) **computer-aided design space exploration**: it is not only possible to quickly verify the performance of the algorithm on various platforms and hardware, but the compiler system can also give feedback to the user about certain design choices (e.g. dimensioning, memory usage, memory transfer bandwidth etc.), 2) **integration of domain-specific optimization techniques**: often different calculation paths may lead to the same result; the compiler can inform the user about possible algorithmic design decisions. 3) **distributed image/video processing algorithms**. With the current programming framework, this is rather trivial to accomplish. However, improved planning and scheduling systems, taking several video streams into account, may be employed.

V. CONCLUSION AND FUTURE DEVELOPMENTS

The Quasar framework is aimed at supporting and extending the research in the domain of image and video processing. The domain-specific language allows researchers to focus on design aspects of the algorithms, rather than on implementation issues. The design methodology has successfully been used within several projects of the IPI research group of Ghent University, including iMinds ASPRO+, GIPA, MMIQQA projects, the IWT SBO Chameleon project and the European Catrene ICAF project. In one of our projects, we have implemented the same algorithm using both Quasar and `CUDA` (by a different researcher), and even though the computational time was nearly identical in both cases, the Quasar version was obtained within one week, compared to three months of development time for `CUDA`.

Currently, the framework is in a research phase, in which several Ph.D. students and postdocs at Ghent University are testing the tools in different image processing application domains (e.g. 3D reconstruction, registration, computer vision, medical image reconstruction). Screenshots and videos are available at http://telin.ugent.be/~bgoossen/quasar/icip2014.

VI. ACKNOWLEDGMENTS

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

VII. REFERENCES

[1] NVidia, "NVidia CUDA Compute Unified Device Architecture," 2007. [Online]. Available: http://www.nvidia.com

[2] *The OpenCL Specification 1.2*, Khronos OpenCL working group Std., 2011. [Online]. Available: http://www.khronos.org/registry/cl

[3] Microsoft, "DirectCompute," 2009. [Online]. Available: http://msdn.com/directx

[4] 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 Transactions on Graphics*, vol. 31, no. 4, 2012.

[5] D. Mikushin and N. Likhogrud, "KernelGen - A Toolchain for Automatic GPU-centric Applications Porting," in *Supercomputing Conference*, Salt Lake City, Utah, USA, Nov. 10-16 2012.

[6] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch, "Rootbeer: Seamlessly using GPUs from Java," in *14th IEEE Int. Conf on High Performance Computing and Communications (HPCC-2012)*, Liverpool, UK, June 25-27 2012, pp. 375–380.

[7] *OpenACC - Directives for Accelerators*, Std. [Online]. Available: http://www.openacc.org

[8] Microsoft, "C++ AMP." [Online]. Available: http://msdn.microsoft.com/en-us/library/hh265137.aspx

[9] "Mozilla Rust." [Online]. Available: http://www.rust-lang.org