Quasar, a High-level Programming Language and Development Environment for Designing Smart Vision Systems on Embedded Platforms

Bart Goossens, Hiêp Luong, Jan Aelterman and Wilfried Philips
IPI - imec - Ghent University
St. Pietersnieuwstraat 41, 9000 Ghent, Belgium
Email: bart.goossens@ugent.be

***Abstract*—We present Quasar, a new programming framework that handles many complex aspects in the design of smart vision systems on embedded platforms, such as parallelization, data flow management, scheduling and load balancing. Quasar, as a high-level programming language, is nearly hardware-agnostic, has a low barrier of entry and is therefore well suited for algorithm design and rapid prototyping. Through several benchmarks and application use cases we demonstrate that programs written in Quasar have a performance that is on a par with (or better than) hand-tuned CUDA and OpenACC code while the development requires much less time and is future-proof.**

I. Introduction

Video processing algorithms in embedded vision systems are expected to process high-resolution video data in real-time (requiring high computational performance) while operating in low-power conditions. In the past years, several embedded platforms have been developed to enable new vision applications (e.g., NVIDIA Tegra, Qualcomm Snapdragon, ...). Despite the promising capabilities of these systems, the programmability of these platform is still a major concern: the programmer does not only need to have in-depth knowledge of the platform (e.g., the GPU hardware) but also the development efforts are quite high, especially when aiming for heterogeneous execution (i.e., the same piece of code can be executed on both multi-core CPU and GPU while making optimally use of the available hardware). In practice, to port real-time video processing algorithms to these new embedded systems, platform experts are required, but these are rarely application specialists (i.e., they often have limited knowledge of the vision and image processing algorithms).

To ease the programmability, low-level programming languages as OpenCL and CUDA have been complemented with extensions, such as OpenACC [1], OpenMP 4.5 with GPU directives [2], C++AMP [3] and OpenVX [4]. These provide functional portability across a range of devices, but in general, target-specific tuning is required for good performance. Moreover, these extensions lack the necessary tools for dynamic runtime decisions and load distribution.

Quasar, a new programming framework for heterogeneous applications, consists of an easy-to-learn language and an integrated development environment. Quasar jointly leverages the combined computational power of both CPUs and GPUs and offers a lower barrier of entry to heterogeneous programming.

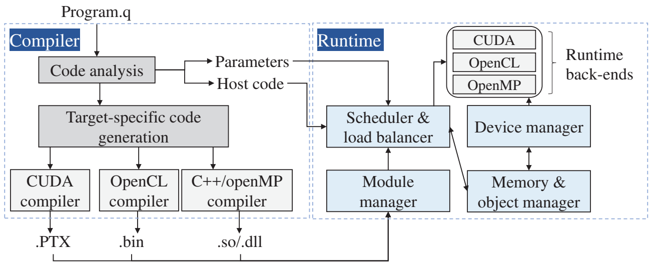


Figure 1. Overview of the Quasar compiler and runtime system.

In fact, Quasar allows for hardware-agnostic programming, where the compiler and runtime system perform memory management, device management, scheduling and load balancing. An overview of the Quasar compiler and runtime system is given in Figure 1. The work flow of the Quasar compiler is as follows: first, the code is analyzed (see Section III), various parameters are extracted (to be used during program execution by the scheduler and load balancer) and both target-specific code (e.g., adjusted to the GPU architecture) and host code ("gluing the target-specific code") is generated. The target-specific code, consisting mostly of *kernel functions*,[1] is then compiled using available back-end compilers, such as proprietary CUDA and OpenCL compilers; and the host-code is either interpreted directly (for debugging purposes) or compiled separately. At runtime, the target-specific binary modules are loaded by the module manager.

The load balancer and scheduler perform a dynamic runtime decision on which device to execute each kernel. The decision combines compile-time parameters as well as runtime parameters (e.g., current load, memory status bits of objects, transfer speeds). In particular, in this paper, we show that a collaborative action of the compiler and runtime is essential to obtain good performance for running vision applications on heterogeneous platforms.

The remainder of this paper is as follows: in Section II we summarize some of the related work. In Section III we give an overview a few common optimization techniques employed by the Quasar compiler, we discuss the synergy

[1]A kernel function is a function that is applied (usually in parallel) to each element of a dataset.

of these techniques with the runtime and we indicate how the interaction between the programmer and the compiler is carried out. In Section IV we discuss some application use cases of Quasar. Benchmarks results are given in Section V. Finally, Section VI concludes this paper.

II. Related work

Several authors address the problem of the programmability of heterogeneous systems. StarPU [5] focuses on runtime techniques to provide a high-level unified execution model for heterogeneous compute devices. In StarPU, the programmer creates parallel tasks for heterogeneous hardware with the possibility to develop and tune the corresponding scheduling algorithms. In contrast to runtime techniques, Amini et al. [6] focus more on compile-time program transformation, eliminating the overhead of dynamic runtime management. Also Leung et al. [7] present a source-to-source compiler, *R-Stream*, to convert high-level, textbook-style algorithms into ANSI C that can run on multi-GPGPU accelerated systems.

Other authors develop domain-specific language (DSL) approaches within general purpose programming (GPP) languages (e.g., C++): Diderot [8] is a parallel DSL for biomedical image analysis and visualization. Halide [9] provides a technique for computational imaging techniques to separate the algorithm from the schedule. With this approach algorithm designers can focus on the algorithms while hardware specialists (assisted with auto-tuning tools) optimize the schedules. The Forma compiler [10] generates code for automatic memory management, data transfers etc. Dandelion [11] extends the the .Net Language Integrated Query (Linq) to a unified programming model for heterogeneous systems. HIPAcc [12] uses domain-specific abstractions to map algorithms through source-to-source compilation to various architectures, taking advantage of different types of parallelism. Finally, other authors add GPU programming support to recent GPP languages. Examples are Rust [13] and Julia [14].

Many of the above works employ compile-time optimization techniques where the runtime system is relatively lightweight. In addition, in [15], several optimization techniques are proposed to improve the efficiency and scalability of distributed-memory task-parallel programs expressed in a high-level programming language (called Swift/T). The authors find many opportunities for cross-layer optimization between the compiler and distributed runtime. In this sense, their work is closely related to our work, although the focus of Swift/T lies more on running external functions written in other languages.

III. Combined compile-time and runtime techniques

In this section, we present two compile-time techniques that are used for generating efficient low-level code from user-supplied Quasar code and we explain how the runtime supports these techniques. The first technique, *reductions*, is a generalization of function overloading to arbitrary expressions. The second technique converts matrix and vector expressions to efficient kernels that can in subsequent steps be fused (*kernel fusion*) or split (*kernel fission*).

A. Reductions

Reductions (also known as rewriting rules [16]) enable library writers to provide efficient implementations for common expressions (e.g., matrix-vector operations), but also allow the compiler to decide between different optimization strategies.[2] In Quasar, reductions are specified as follows:

```
reduction (a1,a2,...,aN) → pattern(a1,a2,...,aN) =
    replacement(a1,a2,...,aN) where conditions(
    a1,a2,...,aN)
```

A reduction consists of a set of bound variables (`arg1`, `arg2`, ..., `argN`), a pattern expression describing the expression to be matched, a replacement expression and a condition expression. The Quasar compiler matches every expression in an input program against all available reductions (note that this can be done very efficiently, see [19]) and evaluates the conditions corresponding to the obtained matches. When the conditions cannot be statically evaluated at compile-time evaluation (e.g., when insufficient information is available on some of the parameters), the evaluation is deferred until runtime. Then, a reduction resolution strategy is used to select the final match. Reductions have several applications:

- Reductions are used to optimize matrix/vector expressions with specialized routines, such as Basic Linear Algebra Subroutines (BLAS) [20].
- Reductions can be used for expression simplification. For example, the following reduction indicates that the real-part of the inverse discrete Fourier transform can alternatively be calculated using a specialized function:

```
    reduction x→real(ifft(x))=realifft(x)
```

Reductions also allow the compiler to “reason” about possible optimization paths. A linear filter could be implemented efficiently in the image domain, except when the filter mask dimensions become too large, the implementation is best performed using Fast Fourier Transforms (FFT). The library writer can provide a generic function for filtering ‘imfilter‘, where compiler then chooses between the two possible implementations, for example, using knowledge of the filter mask size. The compiler also contains the infrastructure to perform inference on the data dimensions (in this case, the filter mask). Then, when a cascade of two filters is provided, both which are implemented in the Fourier domain, the compiler can automatically discard the unnecessary backward and forward FFT transforms.

Summarizing, reductions provide a mechanism to incorporate domain-specific knowledge in the programming language, both on an algorithmic level as on the level of optimization.

[2]Rewriting rules are also common in computer algebra systems, e.g., Mathematica [17], Maple [18], ...

Reductions can also be performed implicitly at runtime, in scenarios when conditions can not be statically evaluated at compile-time.

B. High-level to low-level translation

High-level programming in Quasar consists of manipulating vectors and matrices using operators and functions (in a way similar to MATLAB®). As an array programming language, expressions are quite concise (with advantages similar to mathematical notation) and algorithms are very readable, often like textbook-style algorithms. However, without special compiler support, the use of high-level expressions has a performance cost: the resulting code often exhibits a poor data locality and may indirectly rely on dynamic memory allocation features (which have a significant performance impact on, e.g., a GPU).

As an example, consider the Gaussian density evaluation function in Listing 1. Gaussian function evaluations occur frequently in vision algorithms (such as difference of Gaussians methods, background subtraction, kernel density estimation). The implementation in Listing 1 uses two matrix multiplications per row of the matrix `x` and the result needs to be dynamically allocated, since the dimensions of the vectors and matrices are not known at compile-time (the dimensions may for example depend on data to be read at runtime).

Listing 1. Example Quasar code to evaluate the multivariate Gaussian density function. Here, `mat` is a matrix type and `vec` is a vector type.

```
function y = calc_gaussian_pdf(C : mat, x :
    mat, c : vec)
  y = zeros(size(x,0))
  for m=0..size(x,0)-1
    y[m]=exp(-0.5*x[m,:]*C*transpose(x[m,:]))
  endfor
endfunction
```

The Quasar compiler performs a code analysis step in which constructs that involve "high-level" functions are detected. During the optimization phase, the functions are converted to low-level versions. The following high-level patterns are recognized:

- Aggregation operations (`sum`, `mean`, `prod`, `min`, `max`) of vector/matrix operands and matrix slices (e.g., `sum(A[:,2])`) and aggregation along dimensions (e.g., `sum(A,1)`).
- Matrix resizing functions (`transpose`, `repmat`, `shuffledims`)
- Cumulative functions (`cumsum`, `cumprod`, `cummin`, `cummax`).
- Matrix multiplications and combinations of vector/matrix multiplications.

After conversion to "low-level" code, automatic parallelization can easily identify and parallelize loops. During parallelization, aggregation operations are translated in efficient parallel reduction algorithms (using CUDA shared memory or OpenCL local memory), whereas cumulative functions are converted into parallel prefix sum algorithms. The example from Listing 1 contains a double vector-matrix multiplication. The result obtained after conversion is shown in Listing 2. We obtain a loop nest with three loops: 1) the original loop with iterator `m` and 2) two loops (with iterators `k0` and `k1`) for performing the vector-matrix-vector multiplication in one step. This approach has the immediate advantage that no temporary storage is required for the results of the matrix-vector multiplication, apart from the accumulation variable `out0`, which can be stored and read from the registers.

Listing 2. Loop nest obtained after conversion to low-level code.

```
y=zeros(size(x,0))
for m=0..size(x,0)-1
  out0=0.
  for k0=0..size(C,0)-1
    for k1=0..size(C,1)-1
      out0+=-0.5*x[m,k0]*C[k0,$1]*x[k1,m]
    endfor
  endfor
  y[m]=exp(out0)
endfor
```

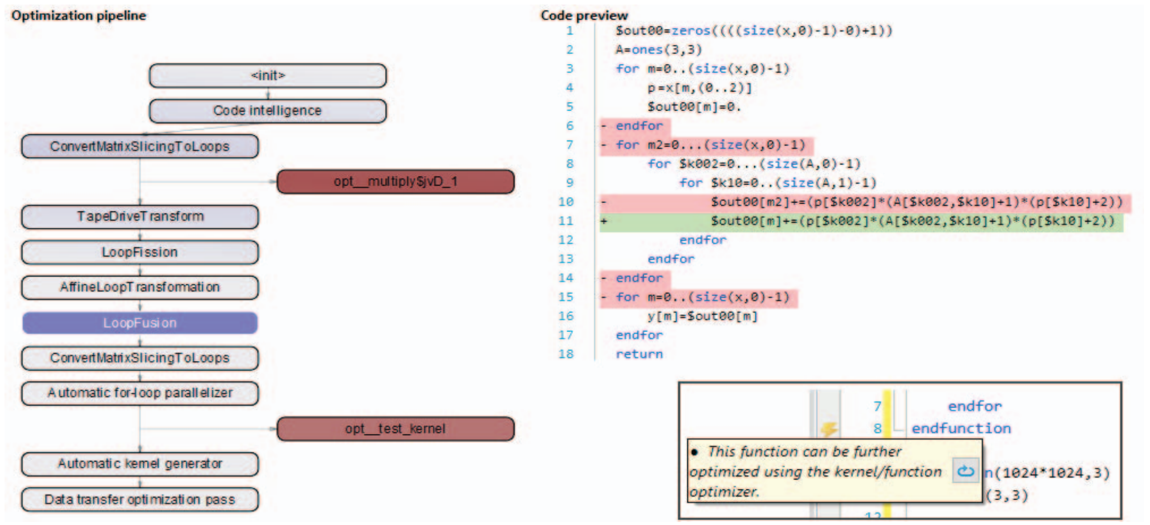
As a next step, array privatization and loop fission are performed [7], causing the intermediate result `out0` to be stored in a vector. Because this vector is allocated from host code (i.e., outside of the loop nest), this has no direct performance penalty.

Listing 3. Loop nest obtained after loop fission (`uninit` allocates uninitialized memory).

```
out0=uninit(size(x,0))
y=zeros(size(x,0))
for m=0..size(x,0)-1
  out0[m]=0.
endfor
for m=0..size(x,0)-1
  for k0=0..size(C,0)-1
    for k1=0..size(C,1)-1
      out0[m]+=-0.5*x[m,k0]*C[k0,k1]*x[k1,m]
    endfor
  endfor
endfor
for m=0..size(x,0)-1
  y[m]=exp(out0[m])
endfor
```

Here, three loops can be parallelized. The middle loop performs a summing aggregation in `out0`, this loop is therefore turned into a parallel reduction algorithm (CUDA/OpenCL back-end) or a reduction pragma `#pragma omp for reduction(+)` (in C++/OpenMP). Furthermore, additional optimizations can be triggered: loop tiling of the middle loop and shared memory caching of the `C` matrix.

Which version is most efficient (Listing 2 or Listing 3) depends on the characteristics of the target devices, as well as the data dimensions. For a CPU, Listing 2 will generally lead to the best performance, whereas for a GPU, Listing 2 is likely to be more efficient *only* when `size(x,0)` is sufficiently large and when `size(C,0)` and `size(C,1)` are small. In



```
  $out00=zeros((((size(x,0)-1)-0)+1))
  A=ones(3,3)
  for m=0..(size(x,0)-1)
      p=x[m,(0..2)]
      $out00[m]=0.
- endfor
- for m2=0...(size(x,0)-1)
      for $k002=0...(size(A,0)-1)
          for $k10=0..(size(A,1)-1)
-             $out00[m2]+=(p[$k002]*(A[$k002,$k10]+1)*(p[$k10]+2))
+             $out00[m]+=(p[$k002]*(A[$k002,$k10]+1)*(p[$k10]+2))
          endfor
      endfor
- endfor
- for m=0..(size(x,0)-1)
      y[m]=$out00[m]
  endfor
  return
```

```
7      endfor
8  endfunction
     n(1024*1024,3)
     (3,3)
```

• *This function can be further optimized using the kernel/function optimizer.*

Figure 2. Quasar transformation pipeline: *(left)* shown are transformation steps that alter the initial code fragment. The brown boxes indicate kernel functions that have automatically been generated. *(right)* the modifications of any transformation (in this case, loop fusion) can be displayed using a graphical *diff*. (*right bottom*) Example of programming hint given by the compiler, which is displayed in the code editor.

some cases, the exact size of `C` can be inferred at compile-time, which causes a specialized algorithm to be generated for which the loop bounds for iterators `k1` and `k2` are constant. On the other hand, when `size(x,0)` is small, more *data parallelism* is exposed through the algorithm from Listing 3.

Therefore, an optimal implementation generates different versions of the algorithm, where the final decision on which version to run is made at runtime, based on parameters that are only known at runtime. The Quasar compiler has an architecture that allows several kernels to be generated for a given loop nest, where the kernel selection is performed at runtime. This also facilitates the generation of target-specific code, where the algorithm of each kernel is adapted to the target architecture. As a way of visual feedback on underlying optimizations, the Quasar compiler shows intermediate implementations (see Figure 2).

To optimize Quasar programs to the target device, dynamic load balancing ensures that the available computation resources are maximally utilized. This requires that all kernels are compiled for multiple targets (e.g., CPU and GPUs) and that memory transfer and management between the different compute devices is handled transparently by the runtime (in some cases, embedded platforms have integrated memory in which the GPU memory is integrated with the CPU memory; which simplifies this task).

To maximize runtime performance, the scheduler and load-balancer need to have a rough estimate of the execution time corresponding to each choice (e.g., running on CPU or on GPU); this information is the result of a collaborative action of both the compiler and runtime: at compile-time, several features are extracted from the kernel functions (such as the number of global memory accesses, the number of registers being used, the maximum amount of GPU shared memory to be used by the kernel...); at runtime, other factors are available (e.g., data dimensions, block dimensions, ...).

IV. Application Use Cases

Several application use cases have already been built using Quasar and several other ones are currently in development. For autonomous vision systems, the following is a selection of the use cases (where the application details can be found in the respective papers):

- *View synthesis* [21], [22]: the goal is to estimate an intermediate RGB view based on a left and a right view with RGB and depth image. A visual example of this technique is given in Figure 3 and Figure 4.
- *Superpixel segmentation* [23]: a sparse image representation based on superpixels improves the processing times of large resolution images.
- *Camera calibration* [24]: automatic camera calibration based on a convolutional neural network.
- *Occupancy map matching for odometry* [25]: a probabilistic occupancy grid model that maps the environment of a moving vehicle in real-time (see Figure 5).

In all of these use cases, a high-level programming approach was adopted, based on high-level expressions complemented by for-loops and kernel functions. This has enabled complex applications to be developed within a short time span (e.g., often about one man month or less per application). Moreover, due to the compiler-time and runtime techniques, these use cases are able to run in real-time on consumer GPUs.

Quasar's compile-time and runtime techniques synergize with its multi-platform support of embedded systems to facilitate multi-platform development: For example, first vision programs are developed within the comfort of a desktop system; in a next step the program can be tested on the embedded architecture (where computational and power constraints limit the complexity of the algorithms). Apart from

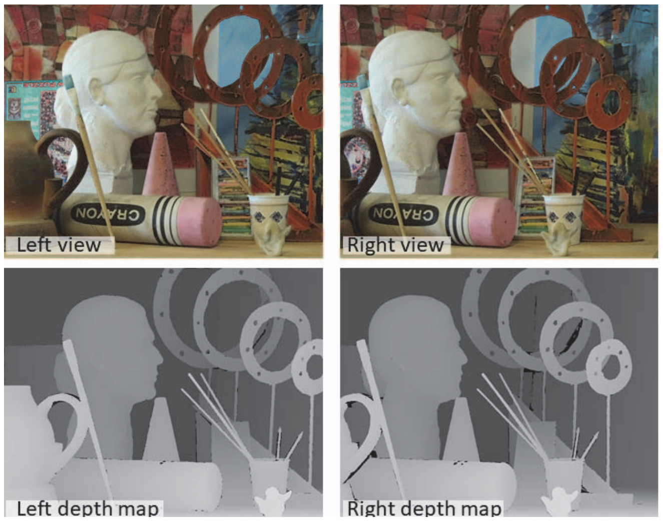


Figure 3. Input RGB and depth images used for view interpolation.

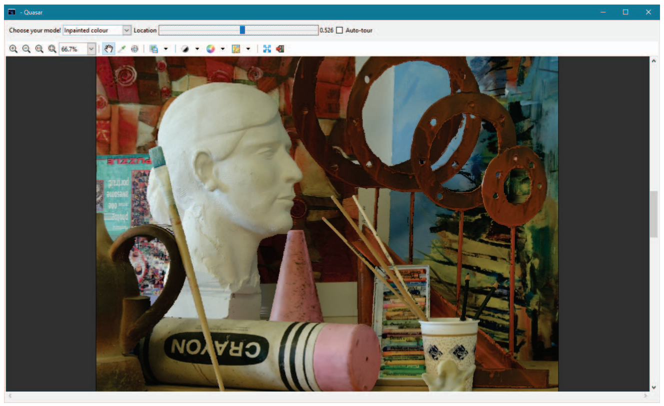


Figure 4. View interpolation: interpolated RGB image. The graphical user interface allows changing the view in real-time (the technique obtains more than 40 frames/second on a NVIDIA Geforce 780M GPU).

architectural optimizations (guided by the compiler feedback), it is necessary to tune some of the algorithm parameters (such as the resolution to be used for depth maps, the number of levels of Gaussian pyramid representation, ...) Therefore, the programmers can focus on the application itself without needing to be bothered too much about the implementation details.

Additionally, the application use cases provide a set of realistic benchmarks that are used to further enhance the Quasar compiler and runtime system. For example, Figure 6 displays the task dependency graph of the runtime execution of the *view interpolation* use case. The runtime system detects implicit task parallelism that can be further maximized by executing tasks out of order (at least, when the dependencies allow it). This technique allows also automatic scaling of Quasar code to multiple CPU cores/GPUs, with no additional programming required.

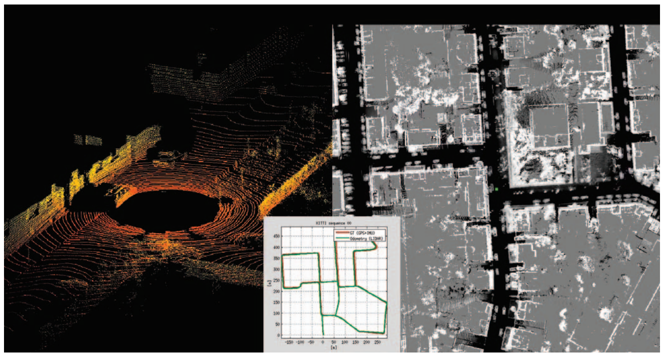


Figure 5. Odometry use case: Point cloud obtained from LiDAR (*left*), calculated occupancy map (*right*) and estimated vehicle trajectory (*middle*).

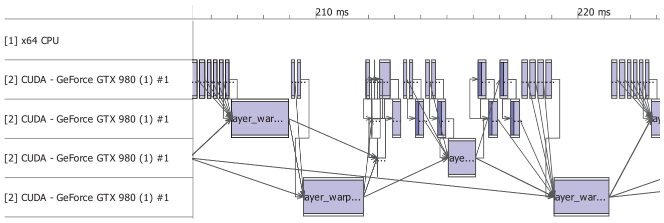

210 ms
220 ms
[1] x64 CPU
[2] CUDA - GeForce GTX 980 (1) #1
[2] CUDA - GeForce GTX 980 (1) #1
[2] CUDA - GeForce GTX 980 (1) #1
[2] CUDA - GeForce GTX 980 (1) #1
ayer_war...
aye...
ayer_warp...
ayer_war...

Figure 6. Timeline with dependency graph obtained from the *view interpolation* use case.

V. Benchmark results

Because the code for the application use cases from Section IV is too extensive to port to other languages for benchmark purposes, we compose a benchmark of single-kernel test programs that incorporate distinct programming patterns that are commonly used within the application use cases. An overview of these test programs, which were implemented in Quasar, CUDA and OpenACC, is given in Table I. All programs were carefully tested, profiled (using NVIDIA nSight and Visual Profiler) and optimized.

In Figure 7, the relative execution time compared to the OpenACC reference execution time are displayed, for an NVIDIA Geforce GTX 980 GPU (the last test program even employs two GPUs). It can be seen that the Quasar execution time compares favorably to OpenACC and CUDA, performing equally well or even better in some cases. Note that CUDA and Quasar perform significantly better on “vectorizable expressions”, “linear filter” and “parallel cumulative sum” due to taking advantage of the shared memory (which OpenACC does not allow). For illustration purposes, the number of lines of code (LOCs) (raw number of lines of code, including empty lines and comments) is also illustrated in Table I. LOCs give an indication of the required programming efforts. Although comparing LOCs between different programming languages and interpreting the results needs to be done carefully, for this test set, on average Quasar programs are 1/4th the length of a CUDA program and 1/2 of the length of an OpenACC program. The difference is mostly due to the amount of device initialization, memory management and memory transfer code.

Table I
OVERVIEW OF SINGLE-KERNEL TEST PROGRAMS USED FOR THE EXPERIMENTAL VALIDATION (LOC=LINES OF CODE).

Test program	Description	LOC (Quasar)	LOC (CUDA)	LOC (OpenACC)
Vectorizable expressions	Kernel function that benefits from high-level to low-level conversion (Subsection III-B)	62	91	22
Partially sequential loop	Kernel function that benefits from affine loop transformations	53	119	23
Memory coalescing	Kernel function that requires dimension shuffling to enable memory coalescing	50	85	21
Jacobi algorithm	Method for determining the solutions of a diagonally dominant system of linear equations	73	160	34
Linear filter	A 7×7 separable Gaussian filter applied to an RGB image	80	147	61
Parallel cumulative sum	Cumulative summing operation along the rows of an RGB image	72	140	37
Radon transform	Parallel beam iterative reconstruction of a computed tomography (CT) image	255	402	149

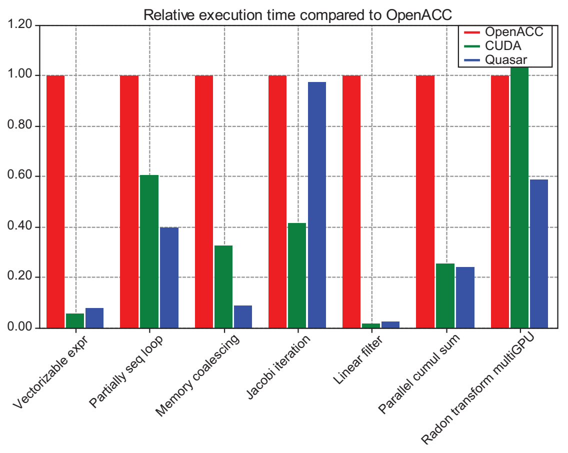


Figure 7. Execution time comparison between OpenACC, CUDA and Quasar for NVidia Geforce GTX 780M.

VI. CONCLUSION

Quasar[3] is a programming environment (with compiler, runtime and IDE) for programming complex heterogeneous applications. The main advantages are: the ease of development (facilitating rapid prototyping of algorithms), the high performance of the generated code (roughly on a par with an expert CUDA implementation) and additionally the future-proofness of the approach (existing code still runs efficiently on future architectures). The core of our approach consists in automatic load balancing decisions steered by both a compiler analysis and runtime parameters which are performed transparently to the programmer. In our future work, we will investigate extending the dynamic runtime decision schemes to select between multiple generated kernel implementations, for cases that compiler analysis remains inconclusive due to missing information.

REFERENCES

[1] "OpenACC - Directives for Accelerators." online: http://www.openacc.org.

[2] O. A. R. Board, "OpenMP application program interface, version 4.5, nov. 2015.".

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

[4] R. Giduthuri and K. Pulli, "OpenVX: a framework for accelerating computer vision," in *SIGGRAPH ASIA 2016 Courses*, p. 14, ACM, 2016.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[6] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static compilation analysis for host-accelerator communication optimization," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 237–251, Springer, 2011.

[7] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, (New York, NY, USA), pp. 51–61, ACM, 2010.

[8] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer, "Diderot: a parallel dsl for image analysis and visualization," in *ACM SIGPLAN notices*, vol. 47, pp. 111–120, ACM, 2012.

[9] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[10] M. Ravishankar, J. Holewinski, and V. Grover, "Forma: A DSL for image processing applications to target GPUs and multi-core CPUs," in *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pp. 109–120, ACM, 2015.

[11] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a compiler and runtime for heterogeneous systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 49–68, ACM, 2013.

[12] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Hipa cc: A domain-specific language and compiler for image processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, 2016.

[13] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, "GPU programming in rust: Implementing high-level abstractions in a systems-level language," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 315–324, IEEE, 2013.

[14] T. Besard, P. Verstraete, and B. De Sutter, "High-level GPU programming in Julia," *arXiv preprint arXiv:1604.03410*, 2016.

[15] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler techniques for massively scalable implicit task parallelism," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pp. 299–310, IEEE, 2014.

[16] B. Felgenhauer, M. Avanzini, and C. Sternagel, "A Haskell library for term rewriting," *arXiv preprint arXiv:1307.2328*, 2013.

[17] S. Wolfram, *Mathematica: a system for doing mathematics by computer*. Addison-Wesley, 1991.

[18] A. Armando and C. Ballarin, "Maple's evaluation process as constraint contextual rewriting," in *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, pp. 32–37, ACM, 2001.

[19] A. Gräf, *Signal processing in the Pure programming language*. na, 2009.

[20] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[21] 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.

[22] B. Goossens, S. Donné, J. Aelterman, J. De Vylder, D. Van Haerenborgh, and W. Philips, "Real-time depth estimation and view interpolation using quasar," *Electronic Imaging*, vol. 2016, no. 19, pp. 1–6, 2016.

[23] S. Donné, J. Aelterman, B. Goossens, and W. Philips, "Fast and robust variational optical flow for high-resolution images using slic superpixels," in *International Conference on Advanced Concepts for Intelligent Vision Systems*, pp. 205–216, Springer, 2015.

[24] S. Donné, J. De Vylder, B. Goossens, and W. Philips, "Mate: Machine learning for adaptive calibration template detection," *Sensors*, vol. 16, no. 11, p. 1858, 2016.

[25] M. Dimitrievski, D. Van Hamme, P. Veelaert, and W. Philips, "Robust matching of occupancy maps for odometry in autonomous vehicles," in *11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2016)*, vol. 3, pp. 626–633, 2016.

[3]Quasar software is available on https://gepura.io/quasar.