Towards Image Processing on Embedded Hardware with Lift

Thomas Kœhler
School of Computing Science
University of Glasgow, UK
t.koehler.1@research.gla.ac.uk

ABSTRACT

Image processing applications are often executed on performance and power constrained embedded hardware. Developing efficient image processing applications on these devices is extremely challenging as current low-level approaches require manual optimization and detailed knowledge of the hardware.

We aim to overcome the challenges of programming and optimizing image processing applications on embedded hardware with the high-level Lift language. The Lift compiler automatically optimizes for the target hardware by applying semantic-preserving rewrite rules. Preliminary results show a significant raise of the abstraction level, and we can expect high performance on embedded hardware.

1 INTRODUCTION

Image processing applications running on embedded hardware are constrained by computing and energy resources, as well as additional requirements such as real-time processing. Optimizing for execution time and power consumption often results in significant improvements that are critical for applications viability. However, developing efficient implementations on the increasingly complex hardware landscape is challenging even for experienced developers.

We argue that Lift [15] is well-suited to address these programming challenges by enabling both high-level abstractions and low-level efficiency. Lift combines a high-level functional language with a rewrite system to define the implementation space.

To effectively use Lift for image processing, we need to be able to express well-known optimizations such as the ones presented in [11], using rewrite rules. To demonstrate this, we investigate the Harris corner (and edge) detector [8] as an initial case study. This is a well established but still relatively simple application, that exposes various implementation choices. Given an input image I (left in fig. 1), it combines point-to-point operations (multiplication, criterion, threshold) and 3×3 convolutions (sobel operators S_x and S_y, gaussian blur G, local maximum) to produce a binary image identifying pixels corresponding to corners. As a starting point, we study the gaussian blur which is an important component of the overall edge detector.

Programming challenges on complex hardware. Hardware architectures are becoming increasingly complex, parallel and heterogeneous to increase performance and energy efficiency [5, 9]. Besides traditional Central Processing Units (CPUs), there are nowadays many-core processors such as Graphics Processing Units (GPUs), reconfigurable hardware such as Field-Programmable Gate Arrays, and specialized hardware such as Digital Signal Processors.

While hardware peak performance is increasing, the achieved software performance is comparatively lacking [10]. The challenges of programming for this complex hardware landscape are leading to unexploited optimization potential. Indeed, optimizing applications by hand using low level code (C, OpenCL, CUDA, OpenMP, etc) requires hardware and domain expertise, and is known to be error-prone, verbose and time consuming. Hardware-specific code transformations are often skillfully applied by hand (e.g. operator fusing, tiling, vectorization) and each new application and target hardware requires going through this process again.

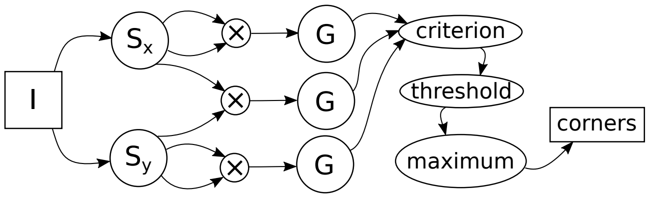

Figure 1: Harris corner detection

These challenges motivate a need for higher-level abstractions to simplify software development and optimization. We strive to increase productivity by clearly separating the abstraction from the implementation details while still achieving high performance with a compiler framework that should be generic across hardware architectures and application domains.

Related Work. Halide [14] is the state of the art in code generation for image processing and is used in production. However, it is domain specific and features ad-hoc internals that are hard to understand from the user-side, and hard to extend. Halide separates algorithm from hardware mapping using a scheduling language, but some optimizations require to change the algorithm. The schedules are mainly written manually, even if automatic scheduling is also being studied. The implementation resulting from a schedule is not obvious, because it depends on the compiler internals.

In comparison, Lift is not yet ready for production but is a promising approach. It is not domain specific and aims to contain no ad-hoc solutions in its core, the rewriting approach is extensible. Lift separates high-level expressions from low-level expressions that encode implementation choices, while allowing expressions in-between. It is able to express both algorithmic and hardware mapping transformations by applying rewrite rules on these expressions. Prior work has shown that it can automatically generate implementations with high performance on various hardware [7, 15]. The implementation resulting from a low-level Lift expression is clear, as it encodes what would be the result of applying a schedule.

There exist many other approaches, some are domain-specific, while others are more generic. High-level representations include higher-order functions, algorithmic skeletons [4] and computational graphs [1]. Optimization techniques include mathematical rewriting [6], polyhedral compilation [13], partial evaluation [12] and refinement of partially defined implementations [3].

2 IMAGE PROCESSING WITH LIFT

Let us overview the Lift language and compilation steps:

$$\begin{tikzcd}\text{high-level} \arrow[r, "\text{rewriting}"] \arrow[loop below] & \text{low-level} \arrow[r, "\text{translation}"] & \text{imperative} \arrow[r, "\text{code generation}"] & \text{C, OpenCL, ..}\end{tikzcd}$$

- *High-Level.* A program is expressed in a functional language with parallel *primitives* such as **map** and **reduce**.
- *Rewriting.* We apply semantics-preserving *rewrite rules* such as map fusion: $\mathbf{map}\ f \triangleright \mathbf{map}\ g \rightarrow \mathbf{map}\ (f \triangleright g)$, where $a \triangleright b = \lambda x.b\ (a\ x)$ first applies a, then b.
- *Low-Level.* Rewriting leads to a functional expression describing an implementation using low-level primitives such as $\mathbf{map_{seq}}$, which implements **map** with a sequential loop.
- *Imperative.* We translate this low-level expression to an imperative intermediate representation (IR).
- *Code Generation.* Finally, we generate code such as C or OpenCL.

We now illustrate this process on the binomial filter.

High-Level. We express the binomial filter in Lift:

$$\begin{gathered}\mathbf{map}\ (\mathbf{slide}\ 3\ 1) \triangleright \mathbf{slide}\ 3\ 1 \triangleright \mathbf{map\ transpose} \triangleright \\ \mathbf{map}\ (\mathbf{map}\ (\lambda nbh.\ \mathrm{dot}\ \left(\mathbf{join}\ \frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}\right)\ (\mathbf{join}\ nbh)))\end{gathered} \tag{1}$$

We start in the first line by creating a 3×3 sliding window using the **slide** primitive which has been introduced to express stencils [7]. Next, we compute the convolution by flattening the neighborhood and the weights to 1D and applying the dot product, defined as $\lambda a\ b.\ \mathbf{zip}\ a\ b \triangleright \mathbf{map} \times_t \triangleright \mathbf{reduce}\ +\ 0$.

Rewriting. We demonstrate rewriting using the register rotation and reduction optimizations described in [11] on (1). For this we first encode the separability property of the binomial filter (it can be decomposed into two 1D convolutions) as a domain-specific rewrite rule (2).

$$\begin{gathered}\lambda nbh.\ \mathrm{dot}\ \left(\mathbf{join}\ \frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}\right)\ (\mathbf{join}\ nbh) \\ \rightarrow \mathbf{transpose} \triangleright \mathbf{map}\ \left(\mathrm{dot}\ \frac{1}{4}[1\ 2\ 1]\right) \triangleright \mathrm{dot}\ \frac{1}{4}[1\ 2\ 1]\end{gathered} \tag{2}$$

Applying this rule on (1) gives us (3).

$$\begin{gathered}\mathbf{map}\ (\mathbf{slide}\ 3\ 1) \triangleright \mathbf{slide}\ 3\ 1 \triangleright \mathbf{map\ transpose} \triangleright \\ \mathbf{map}\ (\mathbf{map}\ \underline{(\mathbf{transpose} \triangleright \mathbf{map}\ \left(\mathrm{dot}\ \frac{1}{4}[1\ 2\ 1]\right) \triangleright \mathrm{dot}\ \frac{1}{4}[1\ 2\ 1]))}\end{gathered} \tag{3}$$

We then use a sequence of generic reorganization rules to obtain (4). It is then trivial to rewrite (4) to the desired low-level expression by selecting low-level primitives where appropriate.

$$\begin{gathered}\mathbf{slide}\ 3\ 1 \triangleright \mathbf{map}\ (\mathbf{transpose} \triangleright \\ \mathbf{map}\ \left(\mathrm{dot}\ \frac{1}{4}[1\ 2\ 1]\right) \triangleright \mathbf{slide}\ 3\ 1 \triangleright \mathbf{map}\ \left(\mathrm{dot}\ \frac{1}{4}[1\ 2\ 1]\right))\end{gathered} \tag{4}$$

Low-Level. The resulting expression is (5). We introduce a new $\mathbf{slide_{seq}}$ primitive that implements sliding window with register rotation, and the remaining high-level primitives will be fused.

$$\begin{gathered}\mathbf{slide}\ 3\ 1 \triangleright \underline{\mathbf{map_{seq}}}\ (\mathbf{transpose} \triangleright \\ \mathbf{map}\ (\underline{\mathrm{dot_{seq}}}\ ..) \triangleright \underline{\mathbf{slide_{seq}}}\ 3\ 1 \triangleright \mathbf{map}\ (\underline{\mathrm{dot_{seq}}}\ ..))\end{gathered} \tag{5}$$

Imperative. We then translate (5) to an imperative IR (shown below) by extending the formalism introduced in [2]. The outer $\mathbf{map_{seq}}$ generates a loop (l.1). We define a translation to imperative for $\mathbf{slide_{seq}}$ and introduce a `newRegRot` imperative construct that creates memory and provides a rotation command (l.2). We generate a prologue to initialize the registers (l.3) before generating a steady-state loop (l.4) which starts by loading the next value and ends by rotating the registers. Reading from the $\mathbf{slide_{seq}}$ input is affected by the fused **slide**, **transpose** and the **map** which performs the vertical reduction. Writing to the $\mathbf{slide_{seq}}$ output is affected by the **map** which performs the horizontal reduction, this creates an accumulator (l.6), initializes it and loops over the sliding window.

```
1  Λh w. λinput. for (h-2) (λy.
2   newRegRot 3 float (λregs rotate.
3    for 2 (λx. take 2 regs.wr @x := take 2 .. @x);
4    for (w-2) (λx.
5     regs.wr @2 := drop 2 .. @x;
6     new float (λacc.
7      acc := 0;
8      for 3 (λi. acc.wr := (..@i) + acc.rd);
9      output@y@x := acc.rd;
10     rotate)))))
```

Code Generation. We then generate code for this imperative IR by extending the mechanism introduced in [2], explaining how to generate code for `newRegRot`. As can be observed, we currently rely on the C compiler to store the created array in registers and to unroll the loops, doing upfront unrolling has shown no benefit in our early experiments.

```
void blur(float* output, int h, int w, float* input) {
  for (int y = 0; y < (h-2); y = (1 + y)) {
    float regs[3];
    for (int x = 0; x < 2; x = (1 + x)) { ... }
    for (int x = 0; x < (w-2); x = (1 + x)) {
      float acc_v = 0.0f;
      acc_v += 0.25f * input[(2+x+(y*w) + (0*w))];
      acc_v += 0.50f * input[(2+x+(y*w) + (1*w))];
      acc_v += 0.25f * input[(2+x+(y*w) + (2*w))];
      regs[2] = acc_v;
      float acc_h = 0.0f;
      acc_h += 0.25f * regs[0];
      acc_h += 0.50f * regs[1];
      acc_h += 0.25f * regs[2];
      output[(x + (-2 * y) + (y * w))] = acc_h;
      regs[0] = regs[1]; regs[1] = regs[2]; }}}
```

3 CONCLUSION AND FUTURE WORK

We showed how Lift can raise the abstraction level to express image processing applications such as the binomial filter, and can be extended to exploit optimisations such as register rotation. We are currently working on benchmarking the generated code on embedded hardware, and will compare it to hand-written and Halide code to demonstrate our ability to generate high-performance code. We intend to continue this work towards *Image Processing on Embedded Hardware with Lift*, starting by completing our harris corner detection case study. Future work may look into expressing further optimisations such as vectorised register rotation, generating GPU code, or exploring the defined implementation space.

ACKNOWLEDGMENTS

I would like to thank the Lift team and my supervisors Michel Steuwer and Phil Trinder.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. http://download.tensorflow.org/paper/whitepaper2015.pdf

[2] Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. 2017. Strategy Preserving Compilation for Parallel Functional Code. *CoRR* abs/1710.08332 (2017). arXiv:1710.08332 http://arxiv.org/abs/1710.08332

[3] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. 2017. Optimization space pruning without regrets. In *CC 2017-26th International Conference on Compiler Construction*. ACM Press, 34–44.

[4] Murray Cole. 2004. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.* 30, 3 (2004), 389 – 406. https://doi.org/10.1016/j.parco.2003.12.002

[5] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 365–376.

[6] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. PÃijschel, J. C. Hoe, and J. M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (Nov 2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[7] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *International Symposium on Code Generation and Optimization (CGO)*.

[8] Christopher G Harris, Mike Stephens, et al. 1988. A combined corner and edge detector.. In *Alvey vision conference*, Vol. 15. Citeseer, 10–5244.

[9] M. Horowitz and W. Dally. 2004. How scaling will change processor architecture. In *2004 IEEE International Solid-State Circuits Conference (IEEE Cat. No.04CH37519)*. 132–133 Vol.1. https://doi.org/10.1109/ISSCC.2004.1332629

[10] W. Jalby, D. Kuck, A. D. Malony, M. Masella, A. Mazouz, and M. Popov. 2018. The Long and Winding Road Toward Efficient High-Performance Computing. *Proc. IEEE* 106, 11 (Nov 2018), 1985–2003. https://doi.org/10.1109/JPROC.2018.2851190

[11] L. Lacassagne, D. Etiemble, A. Hassan-Zahraee, A. Dominguez, and P. Vezolle. 2014. High Level Transforms for SIMD and low-level computer vision algorithms. http://www-soc.lip6.fr/~lacas/Publications/WPMVP14.pdf

[12] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-performance Libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 119 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276489

[13] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 429–443. https://doi.org/10.1145/2786763.2694364

[14] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

[15] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. *International Conference on Functional Programming (ICFP)* (2015).