A Collection of Case Studies of using a Domain-Specific Language for Image Processing Tasks in Parallel Architectures

Victor Oliveira, Roberto A. Lotufo
Department of Computer Engineering and Industrial Automation
School of Electrical and Computer Engineering - Unicamp
Campinas, Brazil
Email: victormatheus@gmail.com, lotufo@unicamp.br

***Abstract*—A development in the field of Domain-Specific Languages (DSL) are programming languages that can target both Multi-Core CPUs and accelerators like GPUs.**

We use Halide, a Domain-Specific Language that is very suited for Image Processing tasks and that can be compiled Just-In-Time, so we can have dynamic development and modularity in a performant way. In order to show both potential and limitations of the Halide language, we implement algorithms that we believe are representatives of key categories in today's Image Processing and Computational Photography.

We compare performance of the Halide implementations with multi-threaded C++ (for multi-core architectures) and OpenCL (for GPUs). We show that once the language suits the problem, the overhead of using it is negligible and there is a good improvement in programmer productivity.

***Keywords*-Domain-specific languages; GPU; Image Processing; Performance;**

I. Introduction

It's a general fact that CPU frequencies have stalled, so recent improvements in CPU architectures have been mostly in the form of vectorization and parallel execution through many cores. Also, in other front, we get GPUs, that used to be very specialized hardware but are now becoming increasingly more general-purpose through languages like CUDA and OpenCL [1]. Unlike in the past, when speedups were gained automatically with frequency scalings in CPUs, this no longer happens, therefore code has to be modified in order to take full advantage of new hardware.

A problem that arises with this technological shift is that if we intend to distribute binaries to an user who has an unknown machine, we are required to have many implementations of the same algorithm, so we can take the best of whatever hardware is present on it. This means having many different codebases, each one tuned for a certain architecture and having little resemblance to each other, as the CPU and GPU architectures are completely different. Being sure that these implementations work right and produce the same results is a hard software engineering problem.

A different but related problem is algorithm optimization for a certain architecture. An experienced programmer can have an intuitive view about how to best map a problem to some hardware, but as architectures become more complex, this turns out to be more akin to art than science and counterintuitive factors may enter into play. For example, we may try to use GPUs to make a certain task faster, but as most discrete GPUs communicate with the main memory through a PCIe bus, we have an extra overhead that is hard to account for. What usually happens is that programmers try a different number of optimizations, but are not able to do an exhaustive search as their time is also valuable.

Image Processing is a field where performance is essential. Images can be large and algorithms can be very complex, specially if we have real-time requirements. Fortunately, a good number of Image Processing algorithms are naturally parallel and can make good use of multi-core CPUs and GPUs, but the aforementioned issues persist.

Domain-Specific Languages (DSL) are programming languages aimed at solving problems in a particular field in computing. The main potential advantage of using DSLs is that, as they have a more limited scope, they can make assumptions that would not be possible in a general-purpose language and therefore, make more aggressive compiler optimizations or have portability across widely different architectures. On the other side, this inherent limitation in the language scope can make some algorithms harder or even impossible to implement. There is a tension between these factors and addressing this trade-off is part of the design of the language.

In this paper we evaluate Halide [2], a DSL for Image Processing that can target both CPUs and GPUs. Its main point is that the algorithm definition is architecture-agnostic and is separated from the machine-dependent execution configuration, called *schedules*.

While the original Halide paper showed that Halide can offer substantial performance improvements over common filter implementations, we would like to know and measure how faster (or slower) it is compared to carefully written and optimized code done with traditional tools usually used by programmers today. Therefore we compare code written in OpenMP using GCC's autovectorization (for CPUs) and OpenCL (for CPUs and GPUs) with their Halide counterparts. In order to make our claims as general as possible, we implemented algorithms from several different common patterns found in Image Processing:

- Color space conversion from RGB to CIELAB [3]
- Motion-Blur
- Bilateral Filter [4]
- Harris & Stephens Corner Detector [5]

In Section II we explain how Halide works and give some examples of Halide schedules. In Section III we describe in depth the filters we have implemented and the optimization choices we have made. In Section IV we show our performance results and our analysis of using Halide. Finally, in Section V we present our conclusion and possible next steps of this work.

II. The Halide Language

DSLs for Image Processing are not a new development, in the commercial options we can cite Apple CoreImage [6] and Adobe PixelBender [7]. In the academic literature, we can see Neon [8], that is an embedded shader language in C#, and Nikola [9], that is also an embedded DSL for imaging in Haskell. All languages also support code generation for both the CPU and GPU, but suffer from the limitation of just supporting pixel-wise operations. General convolutions, for example, are impossible to express in them. As a curiosity, the same approach is being taken in other areas in computing, there are a DSL for evolutionary algorithms [10] and Jet [11], a DSL for fluid simulation, both can run on both CPU and GPU. So this is far from being a trend just in the Image Processing community.

The Halide Language is composed of two parts: An algorithm specification that is done in a pure (free of side-effects) functional language that is architecture-agnostic; and an execution configuration (schedule) that tells how this algorithm is going to run and is very dependent on the target architecture.

Halide also is an embedded DSL, in the sense that it works inside a general-purpose language (C++), this makes easier to just replace critical parts of a traditional codebase with the Halide equivalent.

Here is an small example of a Halide program that computes a 3x3 Box-Blur filter in an image, as this filter is separable, we can run two passes (horizontal and vertical) in the image to produce the final result. We need to “clamp” our input because otherwise we would have an out-of-bounds access in the input when evaluating the result (Halide would also give an error in its bound inference step):

```
UniformImage input(UInt(16), 3);
Var x,y,c;
Func clamped, blur_x, blur_y;

clamped(x,y,c) = input(clamp(x, 0, W-1),
                       clamp(y, 0, H-1), c);

blur_x(x,y,c) = (  clamped(x-1,y,c)
                 + clamped(x  ,y,c)
                 + clamped(x+1,y,c))/3.0f;

blur_y(x,y,c) = (  blur_x(x,y-1,c)
                 + blur_x(x,y  ,c)
                 + blur_x(x,y+1,c))/3.0f;
```

A. Tunning a Halide schedule

A Halide schedule is a declarative language where the configuration of how a function will behave in terms of memory locality, vectorization and multi-threading.

We can see in our Box-Blur filter that there are 3 functions: clamp, blur_x and blur_y, where data dependencies are in the same order respectively. If we don’t specify a schedule, Halide assumes an ’inline’ policy, where for each pixel in the output all necessary dependencies are computed and recomputed without caching anything in memory. So in this policy, each pixel in the output will evaluate 9 pixels in the input and do all calculations for blur_x and blur_y.

We can see that there is a lot of unnecessary recomputation happening for this schedule, which is inefficient. Which leads us to the next schedule.

B. Schedule 1

If we evaluate blur_x for the whole image and reuse these values to compute blur_y, we need 6 memory accesses to the input and less computation. For this we use the ’root’ scheduling policy:

```
blur_x.root();
blur_y.root();
```

Halide schedules basically work on making Loop Scheduling Transformations [12]. If we put the code generated by the Box-Blur algorithm and the schedule above in an imperative language, it is equivalent to:

```
for (y=-1; y<H+1; y++)
  for (x=0; x<W; x++)
    blur_x(x,y) = (clamped(x-1,y) +
                   clamped(x,y)   +
                   clamped(x+1,y))/3.0

for (y=0; y<H; y++)
  for (x=0; x<W; x++)
    blur_y(x,y) = (blur_x(x,y-1) +
                   blur_x(x,y)   +
                   blur_x(x,y+1))/3.0
```

C. Schedule 2

An easy way to improve this schedule is to use threads to compute each stage in parallel:

```
blur_x.root().parallel(y);
blur_y.root().parallel(y);
```

This is equivalent to doing both loops in parallel using many threads, as we can see in Figure 1 it leads to a improvement of almost 4x, which is expected in a 4-core CPU.

D. Schedule 3

Now we can see that schedules can get complicated, but they just reflect the fact that modern architectures are complex.

The most efficient implementation for our CPU needs to take into account:

- split computation of blur_y in y direction in 8 lines blocks.

- use 8-lane vectorization (AVX) in x direction.
- interleave blur_x and blur_y execution (better use of the CPU cache).
- compute blur_y tiles in parallel.

```
blur_y.split(y, y_out, y_in, 8);
blur_y.parallel(y_out);
blur_y.vectorize(x, 8);

blur_x.chunk(y_out, y_in);
blur_x.vectorize(x, 8);
```

It is equivalent to:

```
parallel for (y_out=0; y_out < H; y_out += 8)
{
  for (y_in=-1; y_in < 9; y_in++)
    for (x=0; x < W; x += 8)
      simd8_store(blur_x(x, y_in),
           (simd8_load(clamped(x-8,y_out+y_in)) +
            simd8_load(clamped(x,   y_out+y_in)) +
            simd8_load(clamped(x+8,y_out+y_in))/3.0))

  for (y_in=0; y_in < 8; y_in++)
    for (x=0; x<W;  x+= 8)
      simd8_store(blur_y(x, y_out+y_in),
           (simd8_load(blur_x(x,y_in-1)) +
            simd8_load(blur_x(x,y_in))    +
            simd8_load(blur_x(x,y_in+1))/3.0))
}
```

We can see in Figure 1 that a proper choice of schedules is essential for performance. Although schedules for an architecture can be completely different from another architecture, they allow us to separate architecture-related details and optimizations from the rest, so they mitigate the problem of having to write many versions of the same algorithm. They also make it much easier to discover possible optimizations by trial-and-error.

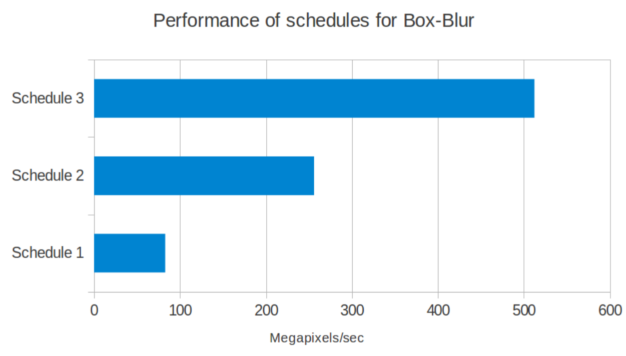

Figure 1. Performance of Box-Blur for each schedule configuration (the higher, the better).

E. Halide Reductions

Halide reductions are composed of two functions: one for initialization and another one for incremental steps that will be done over some range (that is called a Reduction Domain). Basically, if an algorithm can be put in the form of a sequence of stencil operations, that is, matrix computations in which a group of neighboring data elements are combined to calculate a new value, it can be easily expressed in Halide.

Reductions set Halide apart from other DSLs we mentioned because with them we can implement convolutions, projections, downsampling and other image operations that are impossible or hard to express in most previous DSLs we talked about.

Here is an example of reduction where we downsample an image by a factor of 16:

```
// reduction domain
RDom r(0, 16, 0, 16);

Func down;
// initialization
down(x,y) = 0;
// update step
down(x,y) = down(x,y) + clamped(x+r.x-8,
                               y+r.y-8) / 16.0f;
```

If we want to run this operation on the GPU, we just have to change the schedule:

```
down.root().cudaTile(x, y, 16, 16);
down.update().root()
             .cudaTile(x, y, 16, 16);
```

III. Case Studies

In order to evaluate Halide's performance and easiness of use, we decided to implement in C++ and OpenMP, OpenCL and Halide filters with different patterns commonly found in Image Processing so as to make our claims about the language as broad as possible:

- Heavy use of transcendental functions (CIELAB).
- Pixelwise filters (CIELAB).
- Convolutions (Motion-Blur).
- Downsampling (Bilateral Filter).
- 3D and 4D data (Bilateral Filter).
- Complex pipelines (Harris Corner Detector).

These filters were executed in the CPU (for C++ and OpenMP, OpenCL and Halide) and GPU (for OpenCL and Halide).

A. RGB to CIELAB

The CIELAB color space [3] is a device-independent color model that is perceptually uniform, i.e. a difference between values in the color spaces corresponds to the same difference in the human eye color perception. In order to convert from RGB to CIELAB, there are the following steps for each pixel:

1) Gamma Correction:

$$C_{\text{linear}} = \begin{cases} \frac{C_{\text{srgb}}}{12.92}, & C_{\text{srgb}} \leq 0.04045 \\ \left(\frac{C_{\text{srgb}}+a}{1+a}\right)^{2.4}, & \text{otherwise} \end{cases} \quad (1)$$

Where $a = 0.055$ and $C_{\text{linear}} = R, G, B_{\text{linear}}$.

2) Conversion to CIEXYZ:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix} \begin{bmatrix} R_{\text{linear}} \\ G_{\text{linear}} \\ B_{\text{linear}} \end{bmatrix} \tag{2}$$

3) Conversion to CIELAB:

$$\begin{aligned} L^\star &= 116 f(Y/Y_n) - 16 \\ a^\star &= 500(f(X/X_n) - f(Y/Y_n)) \\ b^\star &= 200(f(Y/Y_n) - f(Z/Z_n)) \end{aligned} \tag{3}$$

Where:

$$f(t) = \begin{cases} t^{1/3}, & \text{if } t > (\frac{6}{29})^3 \\ \frac{1}{3}\left(\frac{29}{6}\right)^2 t + \frac{4}{29}, & \text{otherwise} \end{cases} \tag{4}$$

As we can see, converting from the RGB color space to CIELAB is a pixelwise but very expensive operation. Each pixel must be first converted to a gamma-corrected color space (which involves an exponentiation) and the CIELAB formula has a cubic root and a lot of floating-point multiplications.

These factors make this operation very suitable to the GPU as they usually have special hardware to compute transcendental functions such as the ones mentioned. As this is a pixelwise operation, it also has low memory bandwidth requirements.

Nevertheless, there were challenges about porting it to Halide. We had to include in the Halide compiler implementations in NVidia's GPU Assembly (PTX [13]) of mathematical functions like exponentiation that were not available.

B. Motion-Blur

Motion-Blur happens in our everyday when we take a picture with a camera that moves faster than the time to capture the image, so we have an integration of multiple images in the camera sensors along the moving path of the camera.

The filter reproduces this effect by summing over many copies of the input, each one with a slightly different offset. So it is a convolution filter. With it we want to show some problems that arise in image editing and manipulation, where we usually have large images as input and also large convolution masks.

C. Bilateral Filtering

As [4] stated:

> The Bilateral Filter is a non-linear technique that can blur an image while respecting strong edges. Its ability to decompose an image into different scales without causing haloes after modification has made it ubiquitous in computational photography applications such as tone mapping, style transfer, relighting, and denoising.

The Bilateral Filter is a smoothing filter that can be seen as a Gaussian-Blur filter where each output pixel is composed by he weighted average of intensity values of its neighbors. So it is a way to blur the image (removing details) without affecting edges. Formally:

$$BF(p) = \frac{1}{W_p} \sum_q G_{\sigma_s}(||p-q||) G_{\sigma_r}(||I(q) - I(p)||) I(q) \tag{5}$$

While the spatial normal function in this formula can be precomputed, the normal in the color-space domain cannot, so there is need to compute an exponential function for each neighbor pixel, which is expensive.

Fortunately, we have in [14] a fast approximation of the Bilateral Filter (called Bilateral Grid), which is the algorithm we use. The same algorithm has been implemented in the original Halide paper, but was compared against a serial and non-vectorized implementation. We have extended it to 4 channels (a RGBA image, instead of grayscale) and we validate Halide's performance against a traditional implementation of the filter that makes use of multi-threading and vectorization.

The Bilateral Grid algorithm works by subsampling the input image in the spatial and range domain, so we create a three-dimensional grid which is subsequently blurred and interpolated back to the input coordinates to generate the output. So this algorithm is interesting because it explores different data structures in Halide besides the two-dimensional image.

D. Harris Corner Detector

The Harris Corner Detector [5] is an edge and corner detector that is relatively simple and yet produces good results. It has as input a grayscale image and as output a binary image where pixels are black or white depending on if the detector considers them as "important features" in the image. This means that these points are relatively different from their neighborhood and we can expect that the same point will be detected if some affine operation is applied to the original image or in a sequence of frames in a video, for example. So they can be used for Image Registration and Panoramic Photography.

As we can see, the Harris detector is a pipeline of many interleaved stages. In OpenCL, we always try to fuse as many stages as possible together in one kernel, so we decrease the overhead of memory loads and stores in the GPU, the downside of this is that we make code reading more complex, in Halide we have the option of implementing each stage as a function and then using an appropriate schedule to inline functions.

With this algorithm, we intend to see Halide's behavior with complex filter pipelines. Figure 2 shows a high-level view of the stages that compose the algorithm.

IV. Results and Analysis

We have implemented all four filters in C++, OpenCL and Halide and executed the following 5 configurations in a machine with an Intel i5 E5506 CPU (with 4 cores), a Tesla C2050 GPU with a PCIe 3.0 bus and a 4 megapixels test image (2048x2048 and four channels):

1) **C++**: A traditional implementation in C++ using OpenMP for multi-threading and carefully done to make use of GCC's auto-vectorization.

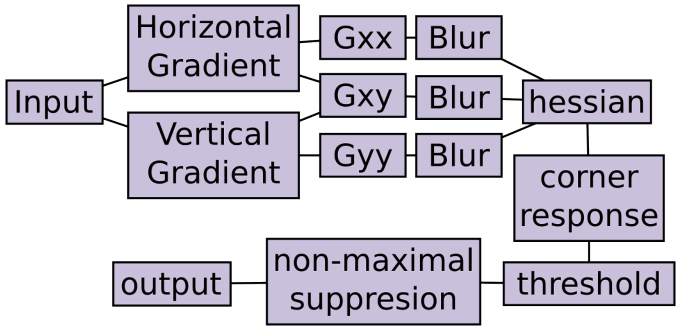

Figure 2. Stages in the Harris Corner Detector.

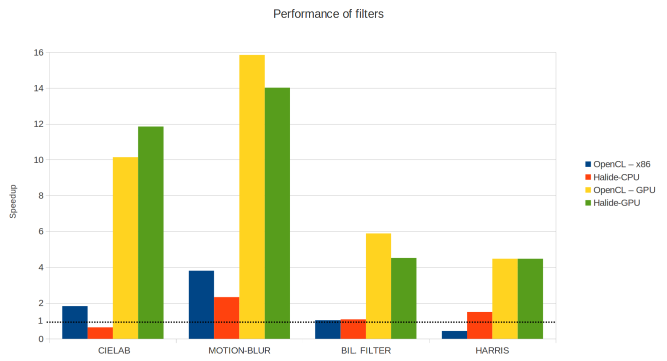

Figure 3. Speedup between OpenCL and Halide implementations over C++.

2) **OpenCL (CPU)**: Code in OpenCL running only in the CPU with the Intel OpenCL compiler.
3) **Halide (CPU)**: Halide implementation with a schedule that configures execution on the CPU.
4) **OpenCL (GPU)**: Same code as Item 2, but running in the GPU with the NVidia OpenCL compiler and drivers.
5) **Halide (GPU)**: Same code as Item 3, but with a schedule that sets execution on the GPU.

Table I show the execution times of these filters and speedups against the traditional C++ implementation. The same speedups are shown in Figure 3.

	C++ and	CPU		GPU	
	OpenMP	OpenCL	Halide	OpenCL	Halide
CIELAB	0.5611 s	0.3075 s	0.8765 s	0.0553 s	0.0473 s
Speedup	1	1.82	0.64	10.14	11.85
Motion-Blur	1.0942 s	0.2879 s	0.4699 s	0.0690 s	0.0780 s
Speedup	1	3.80	2.33	15.85	14.02
Bil.Filter	0.4816 s	0.4639 s	0.4435 s	0.0819 s	0.1067 s
Speedup	1	1.04	1.09	5.88	4.51
Harris Det.	0.3071 s	0.7055 s	0.2050 s	0.0683 s	0.0687 s
Speedup	1	0.44	1.50	4.49	4.47

Table I
PERFORMANCE COMPARISON BETWEEN C++, OPENCL AND HALIDE IMPLEMENTATIONS OF THE FOUR FILTERS WITH RUNNING TIMES AND SPEEDUPS. EACH VALUE IS THE LOWEST EXECUTION TIME AMONG 10 EXECUTIONS.

These numbers show three interesting general patterns:

1) Performance numbers for the GPU are almost similar in OpenCL and Halide, regardless of differences in implementation and execution. In the CPU, except for two outliers in CIELAB and HARRIS, we have almost similar numbers also, but a little more variability than the GPU.
2) The use of GPUs improved performance in our test cases significantly.
3) The traditional C++ and OpenMP implementation almost always was slower than Halide and OpenCL.

A. Analysis

In the GPU, the first thing we can notice is that there is much less difference in running time between traditional implementations and Halide ones. We argue that this is because:

- Image Processing applications in the GPU are usually bandwidth-limited
- GPU superior floating-point processing capabilities compared to the CPU

The rationale behind this is that a good part of the running time in Image Processing filters in the GPU is spent in memory transferences to and from the device. In fact, for the Motion-Blur filter for example, more than 50% of the running time of the filter is spent in these transferences (even using the PCIe 3.0 bus). For reference, without this overhead, the speedup of OpenCL of the Bilateral Filter would be 16.5x instead of 5.88x.

Moreover. the huge amount of floating-point circuitry in GPUs makes differences in generated machine code less relevant.

Of course, if we could amortize this memory transference penalty by a large number of iterations on the GPU, we could get better results. An example of a workflow where this could happen is an Image Editing software where the user has to choose filter parameters and executed repeatedly over the same image, but in this work we do not make this assumption.

These two factors level the playing field for OpenCL and Halide implementations in the GPU.

In the Bilateral Filter, we made extensive use of local and texture memory, features that are impossible to access in Halide right now. Nevertheless, both OpenCL and Halide achieve similar performance (5.88x vs. 4.51x). Suggesting that in the context of Image Processing data does not stay in the GPU long enough for these features to make much difference in processing time.

Moreover, in our validation experiment with the Bilateral Filter, we had less spectacular speedups than the original Halide paper (That had 6x in the CPU and 42x speedups in the GPU) simply because we used a better baseline implementation.

In the CPU, performance numbers are more prone to variation due to implementation choices and compiler optimizations. For example, in CIELAB, Motion-Blur and Bilateral Filter, we achieved better performance in the CPU with OpenCL and Halide because there were many cases where GCC was unable to auto-vectorize some complex expressions, while the Intel OpenCL compiler and Halide (with the appropriate schedule) were able to.

In CIELAB, we have surprising results. Halide in the CPU is significantly slower than the traditional implementation. Inspection of the generated Assembly from both implementations showed that the Halide compiler lacks a Common Subexpression Elimination (CSE) optimization pass, so re-used statements are re-included in the code over and over again. As result, Halide's generated code for CIELAB is 4x larger than ours and a lot of expensive transcendental functions are recalculated without need. Doing CSE in machine code is hard as in this stage we have already lost a lot of information about the algorithm, so having that in Halide would be a great improvement in the language.

We can see that in CIELAB in the GPU with Halide, even with the same aforementioned problems, the GPU took similar amounts of time to run both codes. The duplication of transcendental functions (exponentiation) is not a problem since the GPU has special hardware for handling them. This is a statement to the fact that GPUs have indeed great floating-point processing capabilities and supports our argument that differences in code generation are not that important in the GPU.

In the Harris Corner Detector we have that the OpenCL implementation in the CPU is much slower than the traditional one. We found that this happens because we made extensive use of textures in this filter so we would not need to implement special cases to treat borders, as textures have special hardware in GPUs that clamp memory accesses to be always inside the image. The problem with this is that CPUs do not have special hardware for this and so this clamping behaviour has to be emulated. In fact, the x86 instruction set does not even have a clamp instruction, which contributes to the problem.

This shows that while OpenCL can be trusted with code portability among different platforms, performance is not guaranteed to be optimal without changes in code, something that Halide may offer an interesting solution through the use of schedules.

B. Code Complexity

Table II shows the number of lines of code of each implementation, as an approximation of their complexity. We have not included in Table II the amount of OpenCL code in the host for device initialization, but we do for GPU memory management and kernel calls. Halide numbers include the algorithm and schedules for both CPU and GPU.

We can see that Halide implementations are much smaller than their C++ and OpenCL counterparts. We can specially notice the large complexity of the Bilateral Filter in OpenCL versus Halide's one.

Even considering that the number of lines of code is just a rough approximation to complexity, such a difference makes it clear that it is much easier to have imaging code that works on both CPU and GPU with Halide than the alternatives presented here.

	C++ and OpenMP	OpenCL	Halide
CIELAB	61	77	33
Motion-Blur	70	88	37
Bil.Filter	138	365	45
Harris Det.	196	209	104

Table II
NUMBER OF LINES OF CODE OF EACH IMPLEMENTATION.

Moreover, our experience is that porting code from C++ to Halide is much easier than to OpenCL, as we do not have to worry about GPU memory management and other low-level details and can focus just on the algorithm.

V. CONCLUSION

Even though Halide seems to be a simple language, it is capable of results that can match very optimized implementations in C++ and OpenCL. We, as researchers and developers who have been working with CUDA and OpenCL since almost the beginning of the technology, can say that Halide has had a positive impact in our programming experience for Image Processing. The separation between the specification (which is almost like pseudo-code) and the schedule makes the programmer think first about how to map their problem in parallel constructs which are enforced by Halide's syntax. After writing the specification, we have the ability of being immediately able to test it and make sure it is correct, before spending time improving schedules.

We also think that it is much easier for the Image Processing developer with no experience in code optimization to learn Halide than the current traditional alternatives (C++ with OpenMP and OpenCL). Even if Halide does not achieve more performance than more traditional approaches, its easiness of programming is enough to justify its use.

But as expected, writing good Halide schedules is hard. It requires not only deep knowledge of the underlying architecture, but also knowledge about how Halide works internally and how algorithm schedules are mapped to the generated code, a effort similar to what we we did in Section II with the Box-Blur filter. As we are targeting applications where performance is so essential, having a mental model of Halide internal workings is extremely necessary. This is not a trivial task and there is currently no documentation about Halide internal workings, which complicates it further.

Besides the optimization problems we found in Section III, we also had some problems during development because the Halide compiler is still a research effort and as such, has bugs, specially in non-trivial schedules. Again, understanding how Halides works internally helped immensely, as we were able to solve most problems by analyzing Halide's compiler outputs.

Nevertheless, none of the performance or usability problems we found are inherent to the language and we can reasonably expect that they will be fixed as the language evolves and matures, as documentation is expected to improve.

Also, recent work from the same group that developed Halide shows a way to find schedules automatically through genetic algorithms [15], we have not tried this idea, but it seems promising.

Also, as a contribution of this work, the OpenCL implementation of the Bilateral Filter showed here has been integrated in the open-source image manipulation software GIMP, where it will be used for noise removal and tone-mapping.

As a next step, we are going to push Halide boundaries and try more complex algorithms such as the SIFT image descriptor [16], with the purpose of seeing how far the Halide language can go to express algorithms outside the field of Computational Photography and to use both CPU and GPU to share the workload (Heterogeneous Computing).

References

[1] J. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[2] 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," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 32:1–32:12, Jul. 2012.

[3] I. C. on Illumination, *Recommendations on Uniform Color Spaces, Color-difference Equations, Psychometric Color Terms*, ser. Publication CIE. Bureau Central de la CIE, 1978.

[4] S. Paris, P. Kornprobst, and J. Tumblin, *Bilateral filtering: Theory and applications*. Now Publishers Inc, 2009, vol. 1.

[5] C. Harris and M. Stephens, "A combined corner and edge detector," *Alvey vision conference*, vol. 15, p. 50, 1988.

[6] "Core image programming guide."

[7] "Pixel bender developer's guide."

[8] B. Guenter and D. Nehab, "Neon: A domain-specific programming language for image processing," *Microsoft TechReport MSR-TR-2010-175, Microsoft Research*, 2010.

[9] G. Mainland and G. Morrisett, "Nikola: embedding compiled gpu functions in haskell," *Proceedings of the third ACM Haskell symposium on Haskell*, pp. 67–78, 2010.

[10] S. Dower, "Automatic implementation of evolutionary algorithms on gpus using esdl," pp. 1–8, 2012.

[11] D. Bailey, I. Masters, and M. Warner, "Gpu fluids in production: a compiler approach to parallelism," *ACM SIGGRAPH 2011 Talks*, p. 4, 2011.

[12] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures*. Morgan Kaufmann, 2002.

[13] NVIDIA, "Parallel thread execution," *ISA Version*, vol. 1, 2008.

[14] S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," *Computer Vision-ECCV 2006*, pp. 568–580, 2006.

[15] J. Ragan-Kelley, C. Barnes, A. A., and A. S. Paris S. Durand F., "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *PLDI '13: Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation*, 2013.

[16] D. G. Lowe, "Object recognition from local scale-invariant features," in *ICCV*, 1999, pp. 1150–1157.