Rafael Ravedutti Lucio Machado

High performance code generation using domain specific languages

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Daniel Weingaertner.

Curitiba PR
2017

Resumo

Algoritmos da área de processamento de imagens atualmente devem ser otimizados para diversas arquiteturas presentes no mercado. Porém escrever cada versão otimizada destes algoritmos para estas arquiteturas exige tempo e dinheiro, o que torna a idéia de linguagens específicas de domínio para geração de códigos otimizados atrativa. Usando esta idéia os desenvolvedores se preocupam apenas em escrever a versão alto nível do código sem se preocupar com as otimizações, então o compilador da linguagem se encarrega de gerar a versão baixo nível do código otimizada para diversas arquiteturas estado da arte. O objetivo deste trabalho é explorar o uso e o funcionamento destas linguagens específicas de domínio além de se ter uma idéia de seu uso na prática comparando com os códigos otimizados à mão (hand-tuned), que neste caso é um código que utiliza a biblioteca OpenCV. Usando o algoritmo do detector de bordas Canny, é escrita sua versão em Impala utilizando o framework AnyDSL para geração dos códigos otimizados para as arquiteturas Intel x86 utilizando o conjunto de instruções AVX e o framework CUDA (que permite compilar códigos para serem executados em placas de vídeo NVIDIA). Finalmente o desempenho das versões escrita à mão e gerada pelo framework são comparados para posterior avaliação do uso de linguagens específicas de domínio na prática.

Palavras-chave: canny, anydsl, impala, thorin, linguagens específicas de domínio, processamento de imagens.

Abstract

Image processing algorithms currently must be optimized for several architectures on the market. However writing each optimized version of these algorithms for different architectures demands time and money, what turns the idea of domain specific languages for optimized code generation attractive. Using this idea developers only have to worry about writing the high level version of the code without worrying about the optimization, so the compiler of the language charges itself from generating the optimized low level version of the code for several state of the art architectures. The goal of this work is to exploit the use and behavior of these domain specific languages besides having an idea of their use in practice comparing them with the hand-tuned version of the respective codes, which in this case uses the OpenCV library. Using the Canny edge detector algorithm, its Impala version is written using the AnyDSL framework for the optimized code generation for Intel x86 using AVX instructions and the CUDA framework (which allows to compile codes to be executed in NVIDIA graphic boards). Finally, the performance of both versions are compared for posterior evaluation of the use of domain specific languages in practice.

Keywords: canny, anydsl, impala, thorin, domain specific languages, image processing.

Contents

List of Figures

List of Tables

List of Abbreviations

AMD	Advanced Micro Devices
AST	Abstract Syntax Tree
AVX	Advanced Vector Extensions
CPU	Central Processing Unit
CPS	Continuation Passing Style
CUDA	Compute Unified Device Architecture
PTX	Parallel Thread Execution
DSL	Domain Specific Language
GPGPU	General Purpose computing on Graphical Processing Units
GPU	Graphical Processing Unit
IR	Intermediate Representation
LLVM	Low Level Virtual Machine
MPI	Message Passing Interface
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OpenCV	Open Source Computer Vision Library
PDE	Partial Differential Equation
SIMD	Single Instruction Multiple Data
SPIR	Standard Portable Intermediate Representation
SSA	Static Single Assignment
UFPR	Universidade Federal do Paraná

List of Symbols

α	alpha, first letter from the greek alphabet
β	beta, second letter from the greek alphabet
γ	gamma, third letter from the greek alphabet
ω	omega, last letter from the greek alphabet
π	pi
τ	system response time
θ	light ray incidence angle
P_c	percentage of correct pixels
N_c	number of correct pixels
N_{fp}	number of false positive pixels
N_{fn}	number of false negative pixels

Chapter 1

Introduction

Nowadays many problems on image processing domain require powerful technologies to perform faster and increase their scalability by executing in several processing units. However, to take full advantage of the resources that these technologies provides, specific codes that target each one of them are necessary. Writing these different versions of code for each state of the art architecture available (like x86, CUDA and OpenCL for example) is a time consuming task and requires developers to be familiar with each one of them.

A possibility to solve this problem is the use of domain specific languages, which allows developers to write a high level version of the code that solves a specific problem without worrying about any hardware details. Further, using the code as input, the compiler of the domain specific language must be capable of generating the low level optimized version of the code to many state of the art technologies available.

1.1 Main goal

The main goal of the work is to study solutions on domain specific languages for generating optimized codes to different hardware on the image processing domain, saving the job of writing at least one code for each state of the art architecture available.

1.2 Specific goals

- To study how domain specific languages can help on generating code for several architectures.
- To identify the role of domain specific languages for image processing.
- To implement the Canny Edge Detector algorithm in AnyDSL and compare to OpenCV implementation in different architectures.

Chapter 2

Theoretical Framework

On this chapter is introduced the basic concepts for understanding the work as a big picture. Starting with the domain specific languages concept to understand what is a DSL and what benefits this abstraction can provide to the idea of code generation for multiple platforms. After this, the GPGPU programming model is presented to show possible existing framework that can help the transformation of generic problems written in a DSL to be executed on GPUs. The Canny Edge Detector algorithm used in image processing is also explained as it's used on this work to perform the tests for the explored solution and also the OpenCV library is introduced as it's used as the base for the comparison.

2.1 Domain Specific Languages

A domain specific language (DSL) is a language used for writing programs that solves problems in a determined domain, in contrast with general purpose languages (GPL), which are used to solve problems independent of their domain. For curiosity purposes, not all the DSL have optimization goals, examples are the web languages HTML and CSS.

Some examples of domain specific languages mentioned here are HIPA^{cc}, which is employed to write image processing algorithms, and ExaSlang, a language directed to solve problems by the Multigrid method, a method for solving differential equations in several levels.

A domain specific language may have its exclusive syntax and semantics or be embedded in another existing language (so the DSL developer doesn't need to write the lexical analyzer, parser and the semantic analyzer). Examples are the HIPA^{cc} language, embedded in the C++ language and the ExaSlang, a language which isn't embedded in another language.

The domain specific languages studied in this document have the purpose of creating high level abstractions so the developers don't need to know the target architecture details. This produces standard codes that may then be compiled for multiple target architectures, discharging from the programmer the task of writing many versions of the same code to different architectures.

Domain specific languages may help a compiler to perform optimizations because they usually drive the developer to write his programs without detailing too much the program behavior. An example is when the DSL program doesn't specify the strategy to be performed (how to execute the procedure), only the function to be executed (what to execute), this allows the compiler to decide whatever strategy it defines as the better one for the specified hardware.

Some DSLs may also know certain properties at compilation time by limiting it on the development step (such as the image size, for example) and use this properties to perform optimizations that a GPL language couldn't do. Also, some DSL compiler may generate code for very specific architectures and generate better optimizations by knowing the hardware features in detail (HIPA^{cc} uses this approach).

Another advantage of DSLs is that they usually will be more pleasant to the domain experts for writing and reading code as their representation reflects the domain in which these experts are familiar with.

2.2 GPGPU Programming

The GPGPU programming is the practice of using Graphical Processing Units (GPUs) to execute tasks that normally would be performed by conventional CPUs. This usually occurs to tasks that can be written in a SIMD-optimized fashion, the data is then treated by the GPU as if it were an image and the practice may lead to significant gain in performance as it is executed in hundreds of cores available on the GPUs. The most famous frameworks for working with GPGPU Programming are OpenCL and CUDA.

2.2.1 OpenCL

OpenCL is an open standard and a programming framework created by the Khronos Group to perform parallel heterogenous computing on cross-platform and cross-vendor hardware [Tompson and Schlachter, 2012]. OpenCL standard provides a top level abstraction that allows scalability and writing programs that use massive parallel computations on many types of processing units with no deep knowledge required on their hardware architecture. On GPGPU programming, OpenCL focuses specially on NVIDIA and AMD GPUs.

As mentioned by [Tompson and Schlachter, 2012], the OpenCL data model is defined by a host program (which will be executed on the host processing unit) using the OpenCL API to find the available processing units and then manage the parallel computation by sending workload to these units. The processing part that must be performed by these units is written in form of OpenCL "Kernels" using the OpenCL C language to provide a standard.

2.2.2 CUDA

The CUDA is a framework developed by NVIDIA to perform General Purpose computing on CUDA-enabled GPUs, it allows the host program to copy data to the GPU and launch CUDA kernels that are executed on the GPU using the CUDA grids. The CUDA framework extends typical programming languages such as C/C++ and Python to support functions to copy memory between devices as well as for launching kernels with specific grid and blocks configurations. It also provides features to work with GPU's resources such as the different types of memory available.

CUDA uses the *nvcc* compiler to generate the PTX pseudo-assembly language that is translated by the graphics driver to executable binary code.

2.3 Canny Edge Detector

The Canny Edge Detector [Canny, 1986] is a multistage algorithm that detects edges on images developed by John. F. Canny. The three main principles of Canny are the low error rate (identify as many edges as possible), reduction of false edges caused by noise and accurate detection of the center of each border.

The Figure 2.1 shows an example of the Canny Edge Detector, at the left side is shown the original image and in the right side the Canny Edge Detector result.

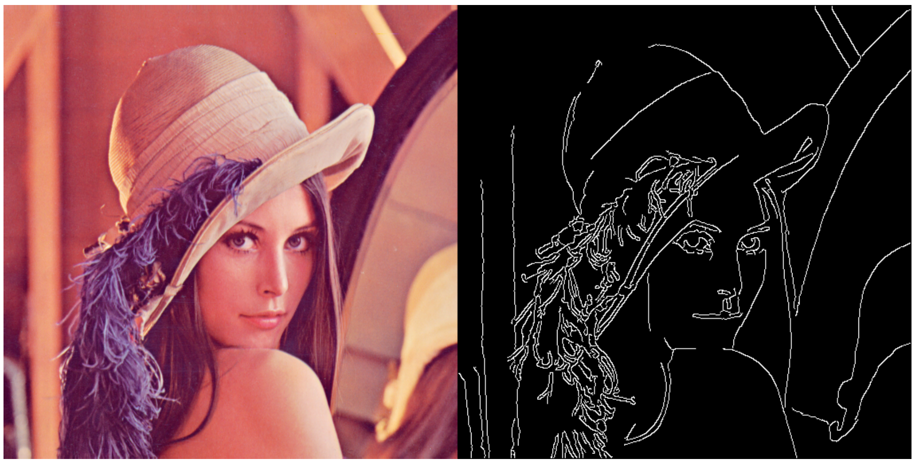


Figure 2.1: Canny Edge Detector example.

Canny's first step is to perform a smoothing operation on the image to reduce noise on the image, this way it's possible to avoid the algorithm to posteriorly detect false borders caused by these noises. For this step the use of the Gaussian filter operator is adopted on this work.

After the noise reduction it's necessary to apply an edge detector operator to detect the edges and the gradients directions on the image, the gradient directions are important for the non maximum suppression step. For this step we use the Sobel operator.

The borders generated by the previous step are still too bold and doesn't precisely tells where exactly are the center of the border (one of the Canny's principles). To solve this problem, a non-maximum suppression algorithm is used to thin the edges and find the center of each border.

After all the steps mentioned, some borders present on the original image are still not present on the final result which compromises the main goal of the Canny. To extract the most part of these edges as possible, edge tracking by hysteresis is performed.

2.3.1 Gaussian Smooth

The Gaussian blur operator is used for blurring the image removing its noise. To transform the image using the Gaussian operator it's necessary to convolve each pixel of the image with the Gaussian function defined as follows:

$$G(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

An example of kernel that may be used to apply the Gaussian smooth is shown below:

$$K = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \end{bmatrix}$$

The Figure 2.2 shows an example of the Gaussian Filter using the same image as Figure 2.1 as the input image.

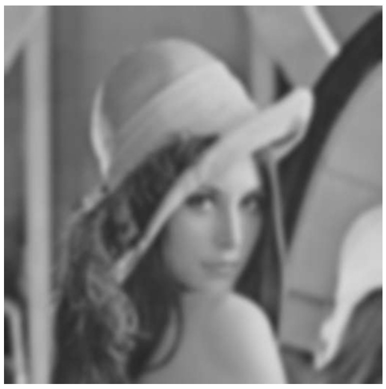


Figure 2.2: Gaussian Filter example.

2.3.2 Sobel

Sobel is used to highlight the edges of the image by giving the intensity and directions of the gradient for each pixel. The first step is to perform a convolution on each pixel using the horizontal and vertical filters for Sobel (respectively G_x and G_y), an example of Sobel kernels are shown below:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

After performing the convolution for both the horizontal and vertical kernels, the intensity (G) and the direction (θ) of the gradient are calculated by the following expressions (using the L2 norm):

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

$$\theta = arctan\left(\frac{Gy}{Gx}\right)$$

The Figure 2.3 shows an example of the Sobel Filter using the same image as Figure 2.1 as the input image.

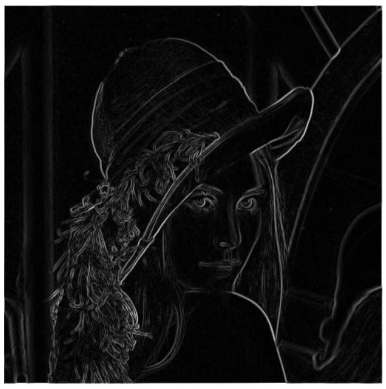


Figure 2.3: Sobel Filter example.

2.3.3 Non-maximum suppression

The edges produced by the Sobel operator are still too bold and we are interested in thin edges showing exactly the point from where the edge emerges. To thin the edges, only the local maximum pixel must be preserved and its neighbors must be suppressed.

We need to compare each pixel with its neighbors in the gradient direction. The direction θ may be rounded to one of the possible angles 0°, 45°, 90° and 135°, for each possibility, the following logic is applied:

1. For $0°$, compare the pixel with its left and right neighbors
2. For $45°$, compare the pixel with its upper-left and bottom-right neighbors
3. For $90°$, compare the pixel with its bottom and upper neighbors
4. For $135°$, compare the pixel with its bottom-left and upper-right neighbors

If the pixel is not the local maximum (i.e. it's lower than one of its compared neighbors), then it's suppressed.

2.3.4 Hysteresis

The final step on the Canny is to define what pixels really are considered as edges. For this, two input parameters are necessary, respectively the low threshold and high threshold.

Each pixel is compared to these threshold values, if the pixel is lower than the low threshold value then it's certainly not an edge and it's suppressed. If the pixel is higher than the high threshold value then it's considered as a strong pixel and certainly is an edge. But if the pixel is between the low threshold and high threshold interval, it's considered as a weak edge.

There are discussions on what to do with the weak pixels, to know if they are or not part of an edge. The criteria adopted by this work is that if a weak pixel is connected to an strong pixel, then it's considered as part of an edge and is set as a strong pixel. This procedure is executed recursively until there are no more weak edges to be updated (i.e. no weak edges connected to strong edges).

2.4 OpenCV

The OpenCV library is an image processing library for solving computer-vision problems. It has several implemented methods for low level image processing providing several high level functionality such as face detection, pedestrian detection, feature matching, and tracking [Pulli et al., 2012].

Since 2010 OpenCV started allowing GPU acceleration on great part of its several implemented algorithms using the CUDA framework. The GPU module allows developers to write codes for CUDA GPUs without having deep knowledge on GPU programming [Pulli et al., 2012].

As OpenCV uses the common approach of writing different versions of its functions to allow GPU acceleration and it's highly acknowledged on the image processing area as a state of the art library, it's used as the base comparison on this work for testing the DSL solution.

Chapter 3

Optimized Code Generation Frameworks

On this chapter are introduced several technologies that uses the DSL solution to generate code for several state of the art technologies available. The purpose of this chapter is not only to show work on DSL for code generation, but also to give an idea of applications that may be developed using the studied methods. It's introduced the HIPA^{cc}, an image processing framework that embeds its language into the C++ language to also generate C++ optimized codes, followed by ExaSlang, DSL for solving problems by the Multigrid method, then AnyDSL, the framework for writing domain specific libraries used for this work's experiments, ending up with Halide, a very promising DSL for generating image processing pipelines.

3.1 HIPA^{cc}

HIPA^{cc} [Membarth et al., 2016] is a framework for generating low-level abstraction codes (source-to-source compiler) for image processing domain that can be executed in GPUs. The codes are written in C++ using a DSL embedded (more specifically, many pseudo-classes that posteriorly will be transformed to more detailed and optimized codes) and are also compiled to C++, generating optimized codes for CUDA, OpenCL and RenderScript frameworks for parallel image processing on its target architectures.

The Listing 3.1 shows a HIPA^{cc} implementation of the Sobel filter convolution as explained in section 2.3.2:

Listing 3.1: HIPA^{cc} Sobel filter kernel

```
class SobelFilter : public Kernel<pixel_t> {
 private:
  Accessor<pixel_t> &input;
  Mask<float> &mask_x;
  Mask<float> &mask_y;

 public:
  SobelFilter(IterationSpace<pixel_t> &iter, Accessor<pixel_t> &input, Mask<float> &
      mask_x, Mask<float> &mask_y) : Kernel(iter), input(input), mask_x(mask_x),
      mask_y(mask_y) {
    add_accessor(&input);
  }

  void kernel() {
    float sum_x, sum_y;

    sum_x = convolve(mask_x, Reduce::SUM, [&] () -> float {
      return mask_x() * input(mask_x);
    });

    sum_y = convolve(mask_y, Reduce::SUM, [&] () -> float {
      return mask_y() * input(mask_y);
    });

    output() = sqrt(sum_x * sum_x + sum_y * sum_y);
  }
};
```

3.1.1 Compiler and Framework

The majority of the HIPA^{cc} optimizations are performed in the kernels, which are the codes that will be run into the target hardware. The framework has a set of available architectures for code generation. The restriction of the set of architectures allows HIPA^{cc} to exploit the suitable memory layout for the images, the GPU's memory hierarchy and the different kinds of parallelism employed by these architectures to generate hardware-specific optimizations.

The written codes are compiled using Clang and generating an Abstract Syntax Tree (AST). Since the generated tree doesn't allow write operations, it's cloned and then the HIPA^{cc} compiler walks through the cloned tree changing the classes and kernels with the optimized codes and replacing the generic function calls by runtime function calls related to the target architecture. For each subclass inherited from the Kernel class, it's generated a file with the optimized version of its kernel subroutine, an example is the *SobelFilter* class in the listing 3.1.

The Figure 3.1 [Membarth et al., 2016] shows the compilation flow for the HIPA^{cc}, the code written in a C++ embedded DSL is transformed to the target architecture language and together with the runtime libraries available runs inside the specified hardware.

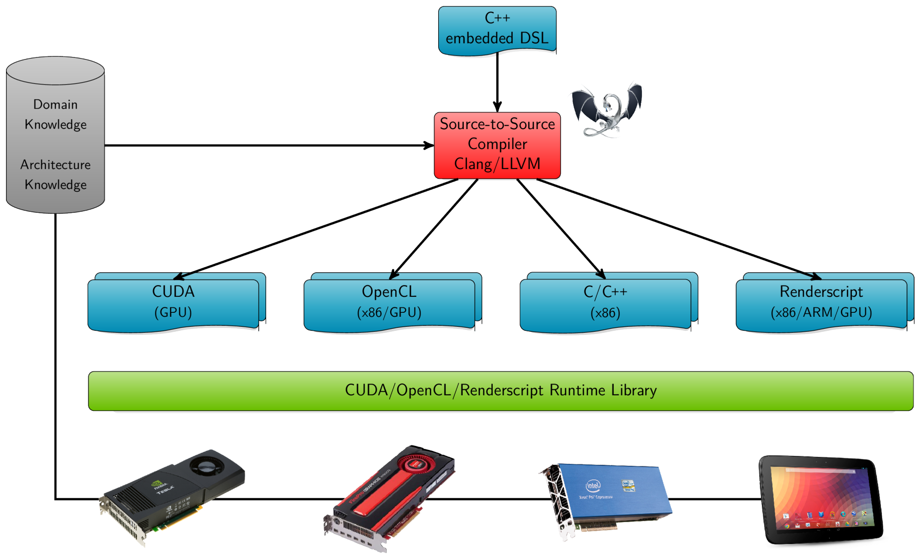


Figure 3.1: HIPA^{cc} compilation flow.

3.1.2 Boundary Handling

Image operations are usually delimited by their boundaries and they must be verified at running time. The cost for doing all the boundary comparisons at all the points of the image is expensive, so this practice must be avoided.

HIPA^{cc} solves this problem by generating nine variants of the operation to be executed at different regions of the image (borders, center and the regions between its borders). In each one of these variants a different boundary checking is performed.

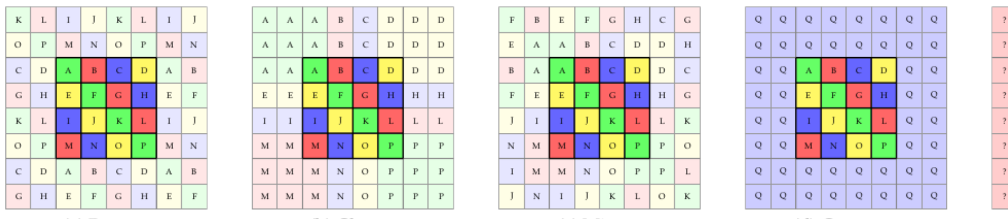
Based on the region that it will be executed, it's easy to deduce which variant will execute which boundary checking. For example, the variant at the top left region will perform boundary checking for left and top boundaries, the center variant won't perform any boundary checking and the center right variant will perform boundary checking only for the right boundary.

Instead of yielding several small kernels (one kernel per segment) in the generated code, HIPA^{cc} chooses to yield a single big kernel that contains all the variants for boundary checking. The reason for this decision is to prevent the overhead caused by launching multiple kernels which could compromise significant performance.

3.1.3 Experimental Results

The following tests from [Membarth et al., 2016] were performed using the Gaussian blur filter on an image of 4096x4096 pixels using a filter of 5x5. The comparisons were made using the RapidMind framework and the NPP library with CUDA, the Halide DSL and the OpenCV library for both CUDA and OpenCL. The naive implementation of HIPA^{cc} uses only global and constant memory.

Tests were made for different types of boundary modes as showing in figure 3.2 [Membarth et al., 2016]: *Repeat* performs the module on the accessed position using the image dimensions. *Clamp* returns the value of the closest valid pixel on the accessed position. *Mirror* returns the value of the symmetrical valid position for the accessed position. Finally, *Constant* uses an arbitrary constant for return on out-of-boundary accesses and *Undefined* returns and undefined value for the pixel.



(a) Repeat. (b) Clamp. (c) Mirror. (d) Constant. (e) Undefined.

Figure 3.2: Boundary modes for HIPA^{cc}.

The Table 3.1 shows the results in *ms* from the CUDA version on a Tesla K20 graphic board. After it, the Table 3.2 shows the results in *ms* from the OpenCL version using a Radeon R9 290X graphic board. Finally, the Table 3.3 shows the results in *ms* from a Xeon Phi 7120P hardware also using the OpenCL framework.

Table 3.1: Performance tests for HIPA^{cc} on a Tesla K20 using CUDA

	Undef.	Clamp	Defined	Mirror	Const.
`naive`	crash	3.04	3.14	3.15	3.19
`RapidMind`	5.40	6.00	crash	n/a	5.97
`Halide`	n/a	4.17	n/a	n/a	n/a
`OpenCV`	n/a	2.12	2.15	2.22	2.01
`NPP`	2.40	n/a	n/a	n/a	n/a
`HIPA`^{cc}	1.32	1.38	1.40	1.38	1.39

Table 3.2: Performance tests for HIPA^{cc} on a Radeon R9 290X using OpenCL

	Undef.	Clamp	Defined	Mirror	Const.
`naive`	1.19	1.18	1.19	1.18	1.20
`Halide`	n/a	0.82	n/a	n/a	n/a
`OpenCV`	n/a	0.89	1.43	0.90	0.91
`HIPA`^{cc}	0.67	0.64	0.68	0.67	0.66

Table 3.3: Performance tests for HIPA^{cc} on a Xeon Phi 7120P using OpenCL

	Undef.	Clamp	Defined	Mirror	Const.
`naive`	crash	6.42	5.96	6.34	6.39
`Halide`	n/a	3.94	n/a	n/a	n/a
`OpenCV`	n/a	10.96	14.06	11.13	9.75
`HIPA`^{cc}	crash	3.57	3.78	3.70	3.87

The HIPA^{cc} version shows a performance advantage in relation to all the other compared implementations for both CUDA using a NVIDIA GPU as well as for OpenCL using an AMD GPU and an Intel CPU, including the hand-tuned implementations on OpenCV and NPP libraries. This demonstrates a considerable advantage as HIPA^{cc} also allows a simpler way for writing the Gaussian filter than the hand-tuned implementations.

3.2 ExaSlang

ExaSlang [Schmitt et al., 2014] is an external domain specific language to develop optimized Multigrid code. The goal of ExaSlang is to compile codes in a high-level abstraction specification to produce low-level abstraction optimized code in C++ using OpenMP and MPI to provide parallelism.

3.2.1 Multigrid

Several problems in numerical analysis are solved using iterative methods. An iterative method is a method which perform many iterations with the same operations on an input and the results of these iterations will becoming each time closer to the result of the problem itself. Some examples of iterative algorithms are Gauss Seidel method and Jacobi, whose purpose is the resolution of linear systems of equations.

The Multigrid method solves a partial differential equation in several levels by using iterative methods to get closer to the result at each iteration. At each level a different granularity is used and so the time and precision of the obtained result varies, this approximation may lead to performance benefits compared to working on a single level.

The pseudocode at listing 3.2 shows the Multigrid V-cycle model algorithm. Before calling the recursions, a pre-smoothing operation is performed (determined by a number of iterations to be performed at the current grid). After the smoothing and residual calculation are done, the problem is transferred to the coarsest grids. After the coarsests grids calculate an approximation, the current grid prolongates the error (which means that it calculates the error in relation to its precision level) and so an approximation to the result in the current grid is performed.

It's important to highlight the presence of the base on the coarsest level, that's why the algorithm firstly verify if it's in that level and, if is, it solves the problem using an iterative method.

Listing 3.2: The Multigrid V-cycle model

```
if coarsest_level then
  solve $A^h u^h = f^h$ (* exactly or by many smoothing iterations *)

else
  $\bar{u}_h^{(k)} = S^{\nu_1}\left(u_h^{(k)}, A^h, f^h\right)$ (* pre-smoothing *)
  $r^h = f^h - A^h \bar{u}_h^{(k)}$ (* compute residual *)
  $r^H = R r^h$ (* restrict residual
  $e^H = V_H(0, A^H, r^H, \nu_1, \nu_2)$ (* recursion *)
  $e^h = P e^H$ (* prolongate error *)
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ (* coarse grid correction *)
  $u_h^{(k+1)} = S_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A^h, f^h\right)$ (* post-smoothing *)
end
```

3.2.2 Compiler and Framework

The ExaSlang compiler and framework are written in the Scala language, a language that is functional and object-oriented simultaneously, and it runs inside Java Virtual Machine (JVM). One of the reasons for using Scala is the possibility for using Parser Combinators to ease the development of parsers. Another advantage of Scala is that it can be integrated with lots of existing libraries.

For purposes of attending many kind of users/developers, ExaSlang follows a multi-layer approach, on which each layer has a different abstraction level for code writing. On the first layer, for example, the developer only specifies the partial differential equation that he wants to solve and ExaSlang becomes in charge of generating the optimized code for the desired architecture.

As the layer number increases, higher the detailing level that must be expressed by the developer and, consequently, higher the knowledge necessary by the same. The most concrete layer is the ExaSlang 4 (which is called Complete Program Specification), in this layer, many functional programming characteristic elements are used for the program specification. The code at listing 3.3 is an example of ExaSlang 1, which employs the Multigrid iterative method V-cycle:

Listing 3.3: ExaSlang 1 V-cycle

```
Function VCycle @((coarsest + 1) to finest)
() : Unit {
  repeat 3 times { Smoother @current () }
  UpResidual @current ()
  Restriction @current ()
  SetSolution @coarser (0)
  VCycle @coarser ()
  Correction @current ()
  repeat 2 times { Smoother @current () }
}

Function VCycle @coarsest () : Unit {
  /* ... solve directly ... */
}
```

In the example above it's possible to note the presence of many elements which are clearly related to Multigrid domain (e.g. coarser, coarsest, finest). These elements represent references to relative grids (like coarser and current) and absolute grids (like coarsest and finest), making easy for the programmer to express himself and increasing the code condensation and legibility.

A crucial step for yielding the output code are the transformations executed by the compiler. The transformations are represented by an input element and an output element (which can be empty, implying in the input code removal at the output code).

At the higher level of transformations, there are strategies, a strategy is a set of transformations to be performed on the input code in relation to one of the steps of the compilation. When a strategy is performed, all its transformations are applied in transactions (which means that can only be applied one transformation per time), which guarantees the nonocurrence of conflicts between transformations that could lead to code corruption.

The example at 3.4 shows a strategy with two transformations in Scala. The first transformation will replace the references to the "foo" function on the input code with "bar" on the output code. The second transformation will solve the adds between two constant values, this way, if there's the 3 + 5 expression on the input code, it will be converted to 8 on the output code, for example.

Listing 3.4: Strategy with two transformations in Scala

```scala
var s = DefaultStrategy("example strategy")

// rename a certain function
s += Transformation("rename fct", {
      case x : FunctionStatement
        if(x.Name == "foo") => x.name = "bar"; x
    })


// evaluate additions
s += Transformation("eval adds", {
      case AdditionExpression(left : IntegerConstant, right :
          IntegerConstant) =>
        IntegerConstant(left + right)
    })


s.apply // execute transformations sequentially
```

3.3 AnyDSL

AnyDSL is a framework that intends to facilitate the domain specific libraries development. An important feature from AnyDSL is its capability of passing functionality through multiple abstraction levels, which is done by using higher-order functions. AnyDSL's main component is its intermediate representation language Thorin [Leißa et al., 2015] (The High-Order Intermediate Representation), a Continuation Passing Style form language [Appel, 1960]. Besides Thorin, AnyDSL also contains Impala [Membarth et al., 2014], an imperative language that aims to make the front-end implementation easier.

3.3.1 Thorin

Many intermediate representation used nowadays are either SSA (for imperative languages) or CPS (for functional languages). However, SSA representations doesn't natively support some crucial techniques like higher-order functions, for example. On the other hand, CPS representations relies on explicit nested scopes, which needs complex transformations to be successfully performed. The Thorin [Leißa et al., 2015] language attempts to solve these problems, as it's a functional language (therefore supports higher-order functions) and doesn't have explicit nested scopes.

To understand the purpose and benefits of Thorin, firstly is essential to know the concept of higher-order functions. A higher-order function is a function f that attends one of the following rules:

1. Has one or more functions as a parameter.

2. Returns a function as its result.

Otherwise, f is a first-order function.

Based on this definition, how are higher-order functions implemented at low level code?

One method for doing it, applied in multiple imperative languages, is the use of closures, which are data structures that contain a pointer to the function address and references to its free variables (variables that aren't local to the function scope). The example at listings 3.5 and 3.6 demonstrates the conversion of a C++ code using higher-order functions to a lower level representation in C using closures (this process is called closure-conversion):

Listing 3.5: Original C++ code with higher-order functions

```cpp
void foo(unsigned int n, function<void(int)> f) {
  unsigned int i;

  for(i = 0; i < n; ++i) {
    f(i);
  }
}

void bar(unsigned int n) {
  foo(n, [=] (int i) {
    do_something(i, n);
  }
}
```

Listing 3.6: C code after closure-conversion

```c
struct closure_base {
  void (*f)(void *c, int);
};

struct closure {
  closure_base c;
  unsigned int n;
};

void lambda(void *c, unsigned int i) {
  do_something(i, ((struct closure *) c)->n);
}

void foo(unsigned int n, void *c) {
  unsigned int i;

  for(i = 0; i < n; ++i) {
    ((struct closure_base *) c)->f(c, i);
  }
}

void bar(unsigned int n) {
  struct closure c = {{&lambda}, n};
  foo(n, &c);
}
```

Although the closure conversion technique works, it has some disadvantages [Leißa et al., 2015]:

1. Its implementation is language specific (in this case, C) and can hardly be converted for use in another front-end language.

2. The generated code size increases significantly, since it's necessary at least a structure per higher-order function.

3. Some compilers like LLVM attempt to reduce the number of closures using the inlining technique (i.e. calling the function address directly and dissecting the closure structure,

replacing it by isolated variables). However, this strategy fails on using recursive higher-order functions.

For CPS intermediate representations, higher-order functions aren't a problem since closures and free variables already belong to the nature of the Continuation-Passing Style (CPS) model.

The Continuation-Passing Style model is the most common model on intermediate representations for functional languages. In CPS each function receives a continuation as a parameter, these continuations are functions with one parameter. After a function is executed in CPS, it must calls its continuation passing its "return value" as parameter (roughly speaking).

As mentioned before, CPS must naturally allows higher-order functions because every continuation is a higher-order function, this helps AnyDSL idea as higher-order functions are a fundamental part for its model. However, functional intermediate representations also support nested scopes by default, which require complex transformations to maintain its variables bound to its definitions (avoid ambiguity) and to keep the program flow correct.

Thorin doesn't have the problem above because it's a functional IR that doesn't have explicit nested scopes, instead, Thorin uses a graph-based approach to represent variable references where each value is a node and each reference is an edge to the value it refers. The Figure 3.3 shows an example of a factorial function in Thorin compared to other IRs, it leaves clear that Thorin doesn't have nested scopes and also shows the edges in the graph [Leißa et al., 2015].

```
fn fac(n: int) → int {
  if (n ≤ 1)
    return 1;

  r: int = 1;
  for (i: int = 2; i ≤ n; ++i)
    r *= i;

  return r;
}
```

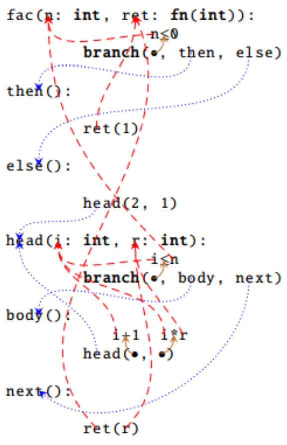
(a) Original program

```
fn fac(n: int) → int {
  branch(n ≤ 1, then, else)
then:
  return 1;
else:
  r0: int = 1;
  i0: int = 2;
head:
  r1 = φ(r0 [else], r2 [body]);
  i1 = φ(i0 [else], i2 [body]);
  branch(i1 ≤ n, body, next)
body:
  r2 = r1 * i1;
  i2 = i1 + 1;
  goto head;
next:
  return r1;
}
```

(b) SSA-form version

```
fac(n: int, ret: int → ⊥):
  let
    then():
      ret(1)
    else():
      letrec
        head(i: int, r: int):
          let
            body():
              head(i + 1, i * r)
            next():
              ret(r)
          in
            branch(i ≤ n, body, next)
      in
        head(2, 1)
  in
    branch(n ≤ 1, then, else)
```

(c) Classic CPS version



```
fac(n: int, ret: fn(int)):
                  n≤0
        branch(•, then, else)
then():
        ret(1)
else():
        head(2, 1)
head(i: int, r: int):
                  i≤n
        branch(•, body, next)
body():
             i+1  i*r
        head(•, •)
next():
        ret(r)
```

(d) Thorin version (blockless)

Figure 3.3: Thorin comparison with other IRs.

3.3.2 Impala

Impala [Membarth et al., 2014] is an extension of Rust that supports both imperative and functional styles, and also uses Thorin as its IR. Impala allows developers to control partial evaluations performed by the compiler using annotations. Also, Impala is able to generate codes to be executed on different target types (like GPUs, for example).

Two valuable features that Impala offers are the detachment of hard-coded implementations (i.e. implementations which behavior relies on fixed data or parameters) and the decoupling of algorithms from its iteration logic (using higher-order functions), which otherwise would difficult code transformation for different architectures [Köster et al., 2014].

The C code at listing 3.7 shows both problems above (is hard-coded and has coupling of operations and iteration logic):

Listing 3.7: C code for applying 1D stencil

```
for(int i = 0; i < size; ++i) {
  out[i] = 0.25f * arr[i - 1] + 0.50f * arr[i] + 0.25f * arr[i
       + 1];
}
```

It is possible to note that the code is hard-coded, since it's necessary to change it for another stencil. Also, its iteration logic is coupled with the operation, since the index of the array is calculated using the iteration variable i. The approach at listing 3.8 can be used for Impala:

Listing 3.8: Impala code for applying 1D stencil

```
let stencil = [ ... ];
field_indices(arr, |i| @{
  out(i) = apply_stencil(arr , stencil , i);
});
```

The code now is decoupled since the higher-order function $field\_indices$ is used and then, the stencil operation is passed as its parameter and can be sent through lower abstraction levels.

Note that now the $field\_indices$ function may vary according to the target architecture the code will be generated. For example, if the code would be compiled for CUDA, the version of $field\_indices$ at listing 3.9 may be used:

Listing 3.9: Impala $field\_indices$ for CUDA

```
fn field_indices(arr : &[float], body : fn(int) -> ()) -> () {
  let dim = (arr.size, 1, 1);
  let block = (128, 1, 1);
  nvvm (dim, block, || {
    let index = nvvm_read_tid_x() + nvvm_read_ntid_x() *
        nvvm_read_ctaid_x();
    body(index);
  });
}
```

Besides its flexibility, a strong point of Impala is the controlling of partial evaluations, using the @ symbol triggers the Impala compiler to perform partial evaluation at a certain point.

Partial evaluations refers to the applying of optimization techniques to the generated code such as loop unrolling, inlining and constant propagation. These transformations are applied by the compiler on walking inside annotated regions in the code. The example at 3.8 shows an example of annotation to perform partial evaluations at the regions traveled using the *field_indices* function.

3.3.3 Framework

It's relevant to note that Impala has built-in functions to generate OpenCL and CUDA source code, as well as SPIR (IR for OpenCL), which allows experts for the target architectures to develop optimized code at lower abstraction levels.

As mentioned previously, AnyDSL passes functionality through multiple abstraction levels using higher-order functions. That means it's possible to use a function at a high abstraction level without declaring it, then linking it at a lower abstraction level where the target architecture is known.

Using the advantage above, the expert for the target architecture can simply write the optimized version of the functions for that architecture. Then, without caring about the compilation details, the domain expert just have to specify what operations to do and the optimizations will be applied to the generated code posteriorly. On the other hand, Thorin becomes responsible for reducing overhead that could be caused by this approach.

3.3.4 Experimental Results

The tests from [Membarth et al., 2014] on the table 3.4 were performed for the NVIDIA GTX 580 and NVIDIA GTX 680 graphic boards using one iteration of the Jacobi iteration method. It compares two CUDA versions as bases, the first is the specialized version that uses the hard-coded stencil and no device-specific optimization and the second is the hand-tuned that benefits from the device-specific resources such as the texture memory for example.

The third implementation is the AnyDSL generic and specialized version that at the moment can't take benefits from device-specific optimizations.

Table 3.4: Performance tests for Jacobi on AnyDSL

	GTX 580	GTX 680
`CUDA (hand-specialized)`	0.33	0.35
`CUDA (hand-tuned)`	0.26	0.23
`Impala (specialized)`	0.32	0.35

The results shows better performance for the AnyDSL version compared to the hand-specialized CUDA version but worse compared to the hand-tuned. As mentioned before, this happens because the AnyDSL version doesn't use all the available resources on the GPUs yet, but this result shows a good approach as Impala version is much more generic than the other implementations. It's also expected that the AnyDSL version reaches the same performance as the hand-tuned version after supporting hardware-specific features.

3.4 Halide

The Halide DSL [Ragan-Kelley et al., 2013] is a domain specific language used to describe image processing pipelines in a simple and functional style. An image processing pipeline may be defined as a sequence of operations to be performed on an image, a local example is the Canny algorithm that have four different operators.

In Halide, every operation on a pixel is written as a function that receives the coordinates as parameters and returns the processed value. The following example from [Ragan-Kelley et al., 2013] shows an unnormalized box filter written in Halide:

Listing 3.10: Unnormalized box filter written using the Halide DSL

```
UniformImage in(UInt(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
Func out(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)
```

The above representation shows the definition of the `blurx` function that applies the unnormalized stencil in the `x` axis followed by the `out` function which applies the unnormalized stencil in both axis using the `blurx` function. The representation is enough to describe the most part of image processing algorithms but Halide also offers `Reduction functions` to express histograms and general convolutions. Also, the boundaries aren't verified in the representation as Halide compute the `guard bands` for optimization and safety purposes.

3.4.1 Scheduling

The defined model above leaves open the scheduling on which the filter steps must be executed. It's noticeable that it makes much easier for the scheduler to perform any kind of strategy by only applying the functions at each point of the image. But still remains the question: what is the best way to schedule the code?

As [Ragan-Kelley et al., 2013] shows, there are three intuitive ways to do that:

- Breadth-first: execute the `blurx` function in the entire image first and then executes the `out` function on the returned values.
- Full fusion: for each pixel, calculates its `blurx` result and in the same iteration perform the `out` function and store the result.

- Sliding window: execute the `blurx` function, saves the returned value into a sliding window to use on the next iteration and executes the `out` function to get the result. The sliding window must have the stencil size.

The Breadth-first strategy performs massive parallel operations as all the points are executed independently but it fails in concerns to locality as it calculates all `blurx` functions for the entire image and only then starts to execute the `out` function (again, for the entire image).

Full fusion also provides high parallelism and enhances locality as both operations are executed in the same iteration for each pixel, but full fusion must calculate the `blurx` function three times for each pixel on the image and so it performs redundant computation.

Finally, the sliding window strategy doesn't perform any redundant work and enhances locality on the algorithm but it introduces a dependency as for each execution of `out` all the calculations of `blurx` for its neighbors must be performed first, which removes significant parallelism from the code.

As each one of these strategies has their drawbacks, what's the best strategy to use? Clearly, all of them are extreme cases as each one only fails in relation to one criterion: locality, redundancy or parallelism. The best strategy then must be a half term that doesn't compromise any of these criteria too much.

Halide provides a model for the strategy definition by choosing the order that the domain must be traversed on each function (which they call `The Domain Order`), the granularity to `compute` each function and the granularity to `store` data to be reused for each function (which they call `The Call Schedule`). The strategy model used for each program is defined before the compilation by an auto-tuner that stochastically searches the space of valid schedules to find one with high performance [Ragan-Kelley et al., 2013].

3.4.2 Compiler

The first step that the compiler does is to generate the loops according to the order defined in the schedule, it performs on the first called function and go through its callees (in the above case, from `out` to `blurx`).

After the generation of the loops the compiler deduces the boundary limits verification for each dimension on the code and then generates the appropriate verification for each function. The bounds inference works on the same order than the first step (from the first called function from to its callees), it computes the bounds using interval analysis based on the bounds evaluated on the caller and the calle indices expressions.

Next, the compiler follows performing optimizations, it goes through the nested loops to find opportunities to perform sliding window optimizations. Then, the compiler performs a flattening on all multidimensional operations changing their multidimensional versions to its single-dimensional equivalent using the computed strides and offsets for each dimension.

Finally, it performs vectorization and unrolling of loops and perform the low-level optimizations according to the backend specified.

3.4.3 Experimental Results

The testes performed at [Ragan-Kelley et al., 2013] uses the algorithms mentioned in the table 3.5, the table also shows the number of functions, how many of these functions are stencils and the complexity of the pipeline structure.

Table 3.5: Example applications for Halide tests

	# functions	# stencils	graph structure
`Blur`	2	2	simple
`Bilateral grid`	7	3	moderate
`Camera pipeline`	32	22	complex
`Local Laplacian filters`	99	85	very complex
`Multi-scale interpolation`	49	47	complex

The Blur uses the example application on the Halide session, Bilateral grid is a fast implementation of the bilateral filter used for smoothing an image without harming its edges. Camera pipeline is an algorithm to transform raw data generated from camera sensors to visible digital images. Multi-scale interpolation works with the image in multiple scales to interpolate pixel data. Finally, Local Laplacian filters also uses a multi-scale approach to enhance local contrast and to tone map images.

Tests were performed for x86 architecture using an Intel Xeon W3520 CPU and for CUDA using a NVIDIA Tesla C2070 GPU. The table 3.6 shows the results for the x86 architecture and the table 3.7 shows the results for CUDA. It's shown the times in *ms* for the Halide tuned version and an expert tuned version written in C, as well as the speedup reached using Halide. The number of lines are also shown to demonstrate that the Halide versions required less effort to be developed, the factor shorter shows the proportion of the expert version lines in relation to the Halide version.

Table 3.6: Performance tests for Halide on x86 architecture

	Halide tuned (ms)	Expert tuned (ms)	Speedup	Lines Halide	Lines Expert	Factor shorter
`Blur`	11	13	1.2x	2	35	18x
`Bilateral grid`	36	158	4.4x	34	122	4x
`Camera pipeline`	14	49	3.4x	123	306	2x
`Interpolate`	32	54	1.7x	21	152	7x
`Local Laplacian`	113	189	1.7x	52	262	5x

The results show a considerable advantage in performance and effort required for Halide versions but it's important to notice that the Interpolate and Local Laplacian versions were compared to the CPU versions as the paper report that no GPU reference for these applications

Table 3.7: Performance tests for Halide on CUDA

	Halide tuned (ms)	Expert tuned (ms)	Speedup	Lines Halide	Lines Expert	Factor shorter
Bilateral grid	8.1	158	4.4x	34	370	11x
Interpolate	9.1	54*	5.9x	21	152*	7x
Local Laplacian	21	189*	9x	52	262*	5x

were available. Nevertheless, Halide still beat all the proposed hand-tuned versions which shows its efficiency and supports the DSL solution on high performace code generation.

3.5 Conclusion

We presented many domain specific languages developed for optimized code generation to show some valuable work on the area and to highlight their contribution on the subject, showing that domain specific languages are being studied to provide solutions on developing optimized code that must be supported by many different kinds of hardware existing in the present moment.

Chapter 4

AnyDSL implementation of Canny

The proposal of this work is to write a Canny version in AnyDSL and compare its results (output and performance) with the state of the art OpenCV implementation of Canny.

The goal is to show that an implemented version in AnyDSL of image processing algorithms tends to be as good as (or even better) than the several hand-tuned versions of the same algorithm written for different platforms, with the advantages that less effort is required to write the filters and it's an easier alternative to multi-platform code generation. We choose AnyDSL because it's a very flexible solution that isn't restricted to a specific domain (we may use AnyDSL for writing domain specific libraries for different domains like Image Processing and Multigrid, for example).

We choose Canny due to its variety of operations (such as Gaussian and Sobel) frequently used in image processing. As each operation has its particularities we show that we can write different types of code in AnyDSL that usually could be a problem for DSLs in general, like the code for Hysteresis for example. After all, we show that AnyDSL is a flexible solution that eases the multi-platform code generation for practically any type of problem.

Using the Gaussian operator we show that the AnyDSL version requires less effort to be written compared to the OpenCV version. The following code shows the generic part of the Gaussian that isn't targeted to any architecture yet (we use a generic version for applying a stencil and pass the gaussian horizontal and vertical filters as parameter):

Listing 4.1: Gaussian filter with no target architectures in Impala

```
/* Apply 2D filter convolution with separable kernels (tiling) */
fn apply_2d_row_convolution(img : image_struct, filter : filter_struct) -> () {
 let anchor = filter.size / 2;

 /* Applies the filter in columns */
 for x, y, arr_in, arr_out, mask in @stencil_iterate(img, filter, false) {
  write(arr_out, y * img.width + x, 0.0);

  for i in @range(-anchor, anchor) {
   write(arr_out, y * img.width + x,
    read(arr_out, y * img.width + x) +
    read(mask, i + anchor) * read(arr_in, y * img.width + x + i)
   );
  }
 }
}

/* Apply 2D filter convolution with separable kernels (tiling) */
fn apply_2d_column_convolution(img : image_struct, filter : filter_struct) -> () {
 let anchor = filter.size / 2;

 /* Applies the filter in rows */
 for x, y, arr_in, arr_out, mask in @stencil_iterate(img, filter, true) {
  write(arr_out, y * img.width + x, 0.0);

  for i in @range(-anchor, anchor) {
   write(arr_out, y * img.width + x,
    read(arr_out, y * img.width + x) +
    read(mask, i + anchor) * read(arr_in, (y + i) * img.width + x)
   );
  }
 }
}
```

As may be noticed, our implementation doesn't check for boundaries in the stencil application. This happens because we define a limitation on the Gaussian filter size. We used the fixed filters with sizes of 5x1 and 1x5 so we can allocate two sentinel borders (extra borders that doesn't have significant content) on the image and don't get any memory access errors without checking the boundaries at every iteration.

Another important point is that we retrieve the coordinates *x* and *y* from the iterate function, as well as the *arr_in*, *arr_out* and *mask* pointers. This happens because these values may differ for different target architectures, Impala allows us to use higher-order functions without worrying about these parameters definition and only at the compilation time these variables will have the proper definition.

After writing the code above, we may use one of the following iterate functions to compile the filter for the CPU or GPU:

Listing 4.2: GPU stencil iteration function

```
/* Iterate for stencils */
fn stencil_iterate(
  img : image_struct,
  filter : filter_struct,
  buffer_to_data : bool,
  body: fn(int, int, Buffer, Buffer, Buffer) -> ()) -> () {
  let acc = cuda_accelerator(0);
  let grid = grid_config(img);
  let block = block_config(img);
  let offset = get_image_offset(img);

  let mut arr_in : Buffer;
  let mut arr_out : Buffer;
  let mask = acc.alloc(filter.size * sizeof[f32]());

  if buffer_to_data {
    arr_in = img.gpu_buffer;
    arr_out = img.gpu_data;
  } else {
    arr_in = img.gpu_data;
    arr_out = img.gpu_buffer;
  }

  copy(filter.data, mask, filter.size * sizeof[f32]());

  with acc.exec(grid, block) @{
    let x = acc.bidx() * acc.bdimx() + acc.tidx();
    let y = acc.bidy() * acc.bdimy() + acc.tidy();

    if x > 1 && x < img.width - 1 && y > 1 && y < img.height - 1 @{
      body(x + offset, y, arr_in, arr_out, mask);
    }
  }

  release(mask);
}
```

Listing 4.3: CPU stencil iteration function

```
fn stencil_iterate(
 img : image_struct,
 filter : filter_struct,
 buffer_to_data : bool,
 body : fn(int, int, Buffer, Buffer, Buffer) -> ()) -> () {
 let mut arr_in : Buffer;
 let mut arr_out : Buffer;

 if buffer_to_data {
   arr_in = img.buffer;
   arr_out = img.data;
 } else {
   arr_in = img.data;
   arr_out = img.buffer;
 }

 for x, y, in block_iterate(img, get_stencil_iterate_block_size()) @{
   body(x, y, arr_in, arr_out, filter.data);
 }
}
```

Each one of these implementations are in separate files and to use them we just need to link the files on the compilation time. We write in these files all the parts of the code that must differ between distinct architectures.

For GPU we get all the configurations for launching the kernel and the $offset$ value to be increased to the x coordinate so we jump the region that refers to the allocated sentinel. Defining the arr_in, arr_out and the $mask$ as the image data allocated on the GPU we assure that the Gaussian works on the GPU region.

The image data at GPU is allocated before the execution of the Canny, we created a function to do the allocations in the GPU file and the same function is created as empty in the CPU file so the compiler doesn't return any error.

In the CPU version above the $block\_iterate$ function is missing, its goal is to perform iteration by blocks on CPU to enhance locality and increase cache utilization for higher performance. Its implementation is shown below:

Listing 4.4: CPU iteration by blocks

```
/* Iterate over image on CPU architectures */
fn block_iterate(img : image_struct, block_size : i32, body : fn(int, int, int) -> ()
    ) -> () {
  let offset = get_image_offset(img);

  for i in parallel(2, 0, (img.height - (block_size + 1)) / block_size) {
    for j in step_range(0, img.width - (block_size + 1), block_size) {
      for k in range(0, block_size - 1) {
        for l in vectorize(8, 4, 0, block_size - 1) @{
          let x = j + l;
          let y = i * block_size + k;

          body(x + offset, y, x);
        }
      }
    }
  }

  for i in range(img.width - block_size, img.width - 1) {
    for j in range(0, img.height - 1) {
      body(i + offset, j, i);
    }
  }

  for i in range(img.height - block_size, img.height - 1) {
    for j in range(0, img.width - (block_size + 1)) {
      body(j + offset, i, j);
    }
  }
}
```

In the *block_iterate* function, we also uses the *parallel* and *vectorize* functions to get CPU optimizations. The *parallel* functions will execute the loop from 0 to $img.height - (block\_size + 1))/block\_size$ using 2 CPU threads (we couldn't found information if it's going to alternate the iterations between the CPUs or share the region for them), this way, the blocks execution will be parallelized in the y axis, which means that the threads will split work between lines of blocks. The *vectorize* function is responsible for applying possible CPU vectorization in the loop region based on the variable l.

The implementation above works for any stencil given as parameter. The procedure used is analogous for Sobel and Non-maximum suppression with the difference that the matrices dx and dy are also included in the function to be used on the target architecture to calculate the gradient directions.

Along with the mentioned optimizations above, we optimized the Sobel operator by unrolling its loop so not all the operations are performed (such as multiplication by one and zero, for example) and no need to store the filter on memory (no memory access need) as the values are present in the instructions.

To optimize the non maximum suppression stage we create a matrix of offset indexes, this way, by rounding the gradient angle to 0°, 45°, 90° or 135° we already perform an association to the indexes 0, 1, 2 or 3. By accessing the offset indexes matrix with the calculated index, it returns the offsets to get the neighbors to be compared with the current pixel, avoiding several comparisons.

For hysteresis we use a stack to save all the weak edges and the procedure stops only when the stack is empty. At each iteration if the current pixel is set as a strong edge, all its neighbors are pushed to the stack if they are marked as weak edges. Also, we remove the need from the performance-cost expensive square root function by working with the square values and using the low and high threshold values as $low\_threshold^2$ and $high\_threshold^2$, respectively.

On comparing the code above to the OpenCV implementation we can clearly notice the significant difference in effort as it has entirely different implementations for each architecture in OpenCV. Only CUDA implementation of Canny (with no Gaussian implemented) in OpenCV has about 500 lines of code and it requires a high level of knowledge in CUDA to implement. Also, the CPU version has about 1100 lines of code and requires good knowledge on programming with AVX/SSE. Our version has 875 lines in total including all the image struct basic methods and the procedures for Canny. Another important detail is that if we want to compile our code for OpenCL, we just need to change the acc attribution using the $opencl\_accelerator$ function instead of $cuda\_accelerator$.

The point on this work is to show that we can get a good approximation on performance with much less effort using domain specific languages comparing to the state of the art technologies that has one version of code written for each architecture available.

As shown above, the AnyDSL implementation doesn't take too much time and knowledge to be written while OpenCV implementation does.

4.1 Discussion

We experienced some problems working with AnyDSL as it's on development stage since the start of this work. One of these problems we found on building it and we couldn't find the solution which only turned possible for us to use it after the framework being update. Also, due to the lack of documentation, we had to see the framework sources and examples to learn how to use certain features of AnyDSL like the functions for executing the code in a GPU, for example. Some code we wrote for GPU resulted in errors during Clang compilation of the generated code which forced us to find a solution without exactly knowing what was happening. By exposing these issues, we also show that AnyDSL may still have a lot more to offer in the future as it's still being developed in the present and that even on a starting stage it already has many useful features to contribute with researches on domain specific languages for optimized code generation.

Chapter 5

Experimental Result

In this chapter we present the methods and materials used to perform the tests for validation and performance of our version, as well as the test results.

5.1 Materials and Methods

To perform the experiments and validate our AnyDSL implementation of the Canny algorithm, we use the Berkeley Segmentation Dataset [Martin et al., 2001] available at [Arbelaez et al., 2007], which contains 100 images with 321x481 and 481x321 dimensions. As these dimensions are too small for performance comparison, we use the same procedure as [Lourenço, 2011] to generate the variations with higher dimensions.

Considering the original dataset images as the B1 dataset, we generate the B2 dataset with the doubled dimensions by repeating the original images vertically, horizontally and diagonally. So, we repeat the same procedure on B2 to generate the B3 dataset, and the same on B3 to generate the B4 dataset. After all we have the following dimensions: 321x481 and 481x321 for B1 dataset, 642x962 and 962x642 for B2 dataset, 1284x1924 and 1924x1284 for B3 dataset and 2568x3848 and 3848x2568 for B4 dataset, all of them containing 100 images each.

The performance tests were performed using an Intel Core 2 Duo E7500 processor with 3MB of L2 cache and 2.93GHz to generate the CPU results, and a GeForce GT 710B graphic board with 192 CUDA cores, 954MHz base clock, 2GB of memory with 1.8 Gbps of speed and a 64-bit DDR3 memory interface for the GPU results. We execute both algorithms for all the images on each dataset and considered the summation of the times as the final result for our tests.

We perform the validation tests to assure that our Canny implementation is correct and the performance tests to check if it has good performance compared to the state of the art OpenCV implementation.

5.2 Validation Tests

To validate the Canny, we compared each pixel of the images generated by both implementations and classified them according to the following specification:

- Correct pixel: pixel is defined as a border in both implementations.
- False negative: pixel is defined as a border only by OpenCV implementation.
- False positive: pixel is defined as a border only by AnyDSL implementation.

Basing on this definition, we calculate the percent of correct pixels using the following equation:

$$P_c = \frac{N_c}{N_c + N_{fp} + N_{fn}}$$

Being N_c the number of correct pixels, N_{fp} the number of false positive pixels and N_{fn} the number of false negative pixels.

As Canny allows many variations and there's no detailed documentation about the OpenCV implementation of Canny, we couldn't reach the exactly same output results as them. Instead, we come to a good approach (almost 90% of the result) and validate by empirical analysis that the borders generated only by our implementation reflect true borders on the original image. The following table shows the percentage of correct pixels for the different datasets B1, B2, B3 and B4:

Table 5.1: Validation results for the AnyDSL implementation

	Correct percentage N_p
`B1 dataset`	0.87
`B2 dataset`	0.87
`B3 dataset`	0.88
`B4 dataset`	0.88

The Figure 5.1 shows an image from the Berkeley Segmentation Dataset and the figure 5.2 shows its results for both AnyDSL and OpenCV implementations, as well as the false positives pixels (pixels set only in AnyDSL result) and false negatives (pixels set only in OpenCV result).

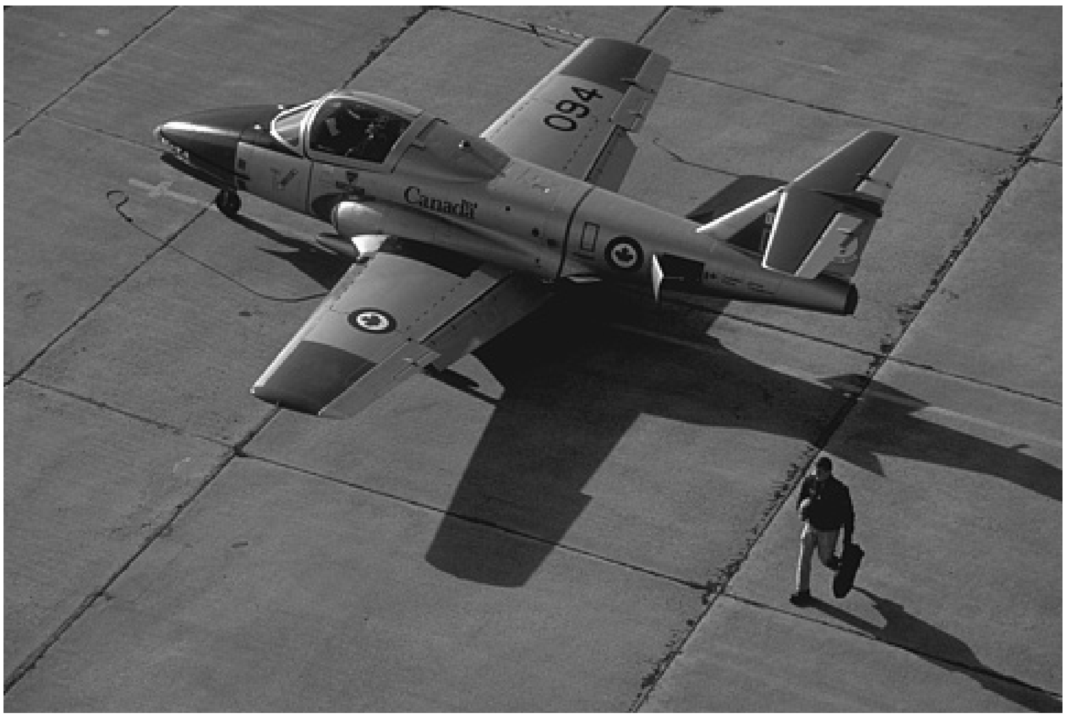
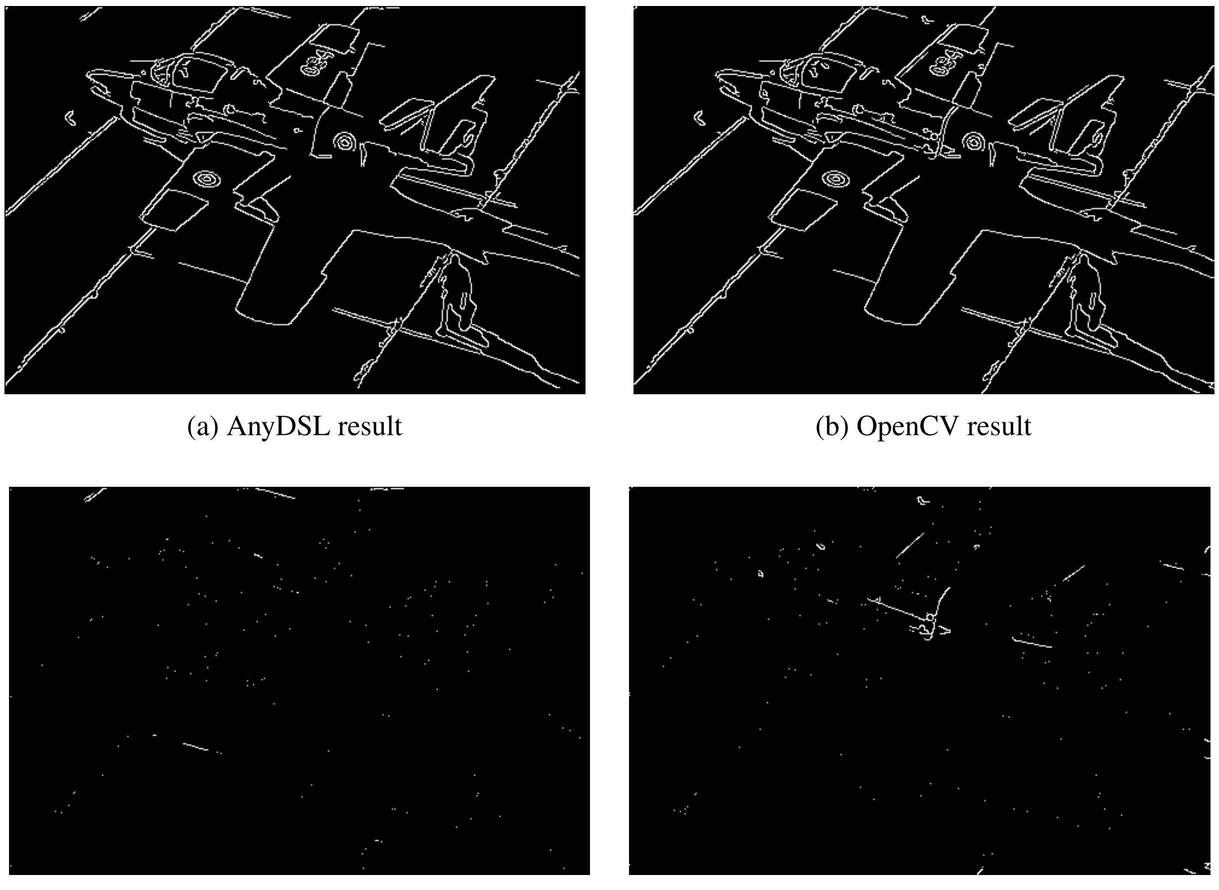


Figure 5.1: Sample image from Berkeley Segmentation Dataset.



(a) AnyDSL result

(b) OpenCV result

(c) False positives

(d) False negatives

Figure 5.2: Comparison between AnyDSL and OpenCV results for figure 5.1, (a) shows result generated by AnyDSL, (b) shows the result for the same image for OpenCV, (c) shows pixels set only in AnyDSL result and (d) shows pixels set only in OpenCV result.

The comparison shows a displacement for some pixels which also caused certain borders to be extended only by one implementation, the contiguous borders found on the false positives and false negatives masks clearly shows this issue.

As may be noticed, although the difference exists in the results, both versions identified the borders in the image with good quality, this is enough to validate the AnyDSL code as an authentic version of the Canny Edge Detector.

5.3 Performance Tests

The Figure 5.3 shows the performance results for CPU with the time in seconds using the *y* axis in logarithmic scale (as the image sizes grow up exponentially). We notice an overhead on OpenCV version during the B1 and B2 datasets, which made our version perform much better for smaller images. Our performance loses only after the tests for B4 dataset, but still doesn't get too far from OpenCV performance.

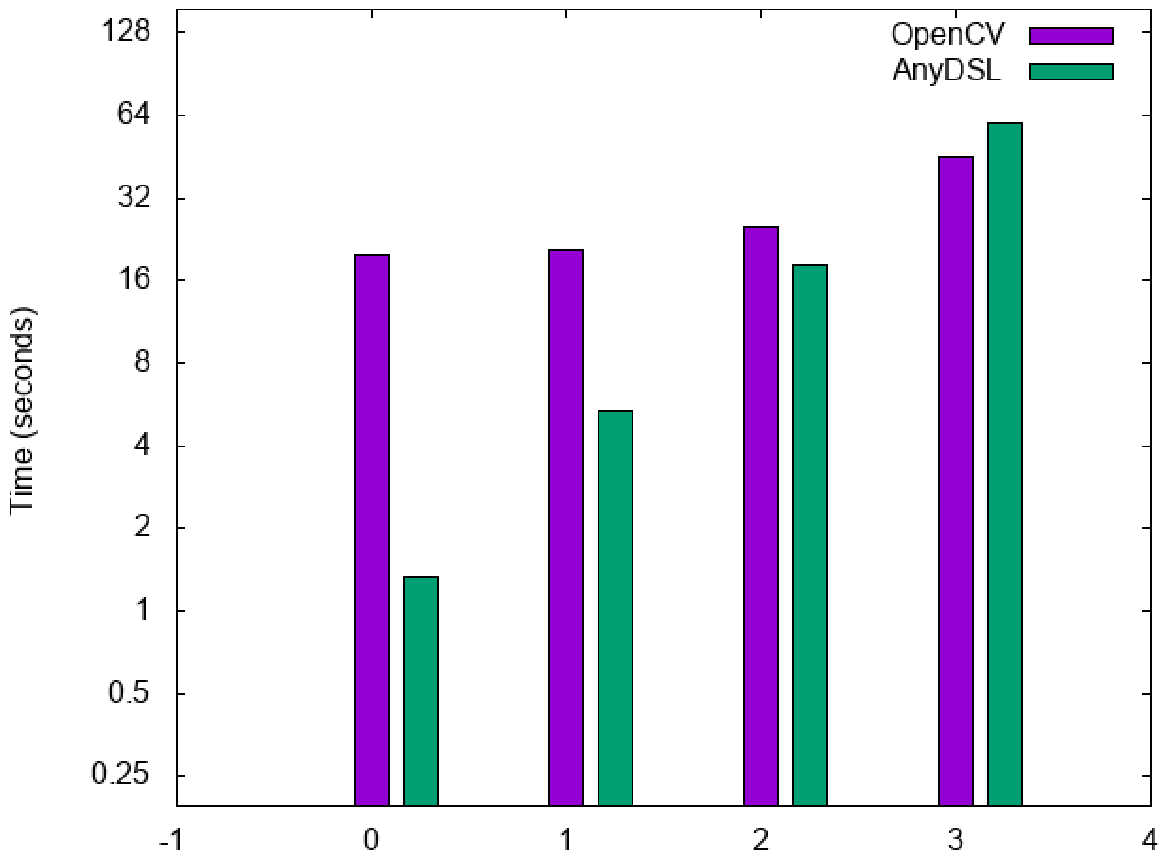


Figure 5.3: Canny results for AnyDSL and OpenCV using CPU.

A possibility to enhance the performance for AnyDSL version in CPU would be to optimize the block sizes in relation to the CPU cache size available and also try to employ some parallelism at Hysteresis stage.

The Figure 5.4 shows the performance results for CUDA in seconds using the *y* axis in linear scale.

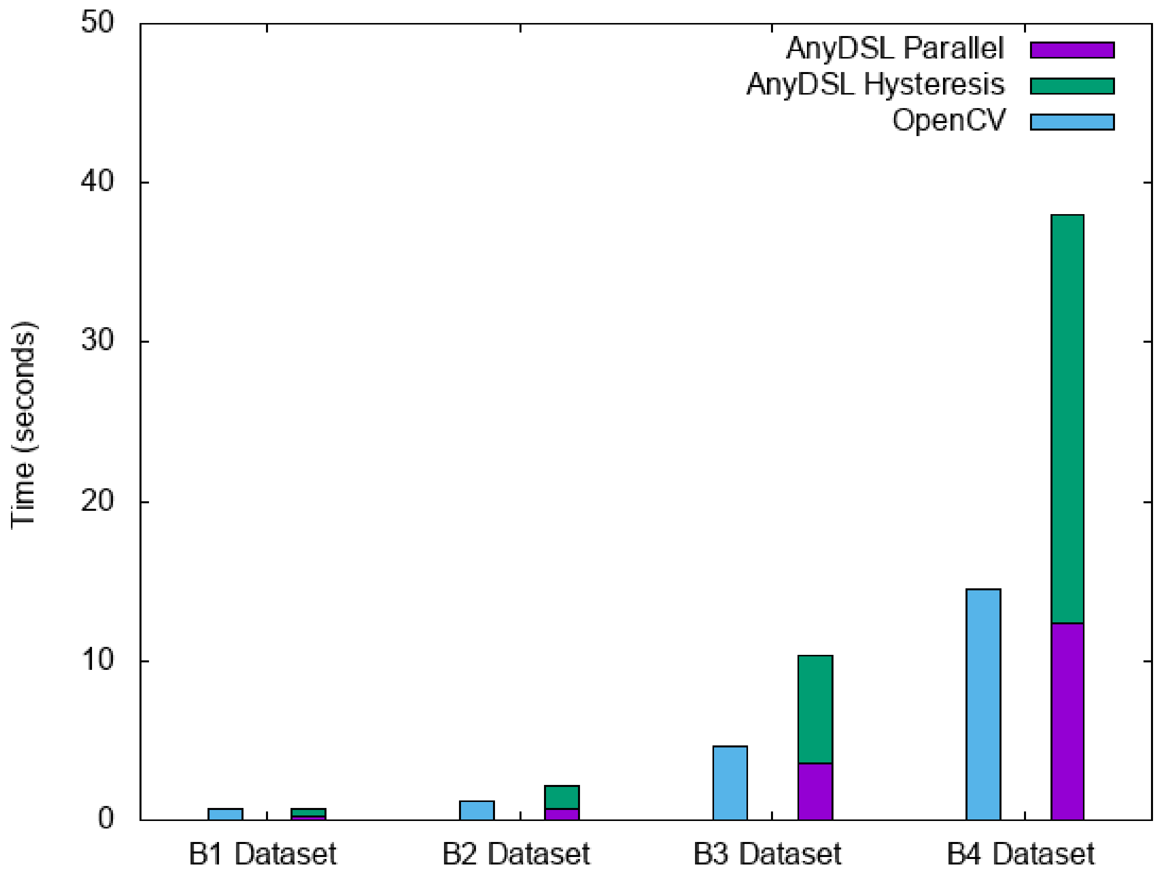


Figure 5.4: Canny results for AnyDSL and OpenCV using CUDA.

A solution to achieve better performance for the AnyDSL algorithm would be to write a parallel version of the Hysteresis as mentioned by [Luo and Duraiswami, 2008], for example. This way, the Hysteresis portion on the GPU results in figure 5.4 would shrink and both versions would have similar performance, even considering that AnyDSL version doesn't use GPU specific resources.

Chapter 6

Conclusion

In this work we studied and presented many solutions using Domain Specific Languages for optimized code generation. The solutions provide an alternative to traditional solutions writing different versions of the same code for each architecture it must support.

The DSLs provide a much better way for writing multi-platform code as writing a single code in a DSL doesn't need any references to the target architectures and requires much less effort compared to several hand-tuned versions of the same code in low level languages. Also, DSLs helps the compiler to find and apply optimizations as it can defines the best strategy for a high level code to execute in a certain hardware.

Another important point is that codes written in DSLs turn to be much more comprehensible to domain experts as they usually will be more familiar to the features and elements these languages provides. The ExaSlang language is a good example of this as it allows the domain experts to write code using elements from the Multigrid domain. The image processing DSLs HIPA^{cc} and Halide also provide much better solutions for writing image processing codes in relation to General Purpose Languages.

We also present a Canny version using the AnyDSL framework for multi-platform code generation, showing that it's easier to write the DSL code even not having deep knowledge about the target architectures it'll be executed. The performance tests also gave a good approach to our version in relation to OpenCV even considering that our version doesn't use a parallel version of Hysteresis and GPU specific-resources, enhancing that it's possible to use DSLs and get good results on performance.

6.1 Future work

Among the possibilities opened by this work, it worth to mention the use of AnyDSL or another DSL framework to generate energy-efficient code instead of performance optimized as mentioned here.

Another possibility is to use AnyDSL to optimize codes that uses limited or fixed parameters like a fixed image size for example, an perform optimizations based on this limitation. Another approach is to use AnyDSL to generate code for a very specific model or configuration of a target hardware like HIPA^{cc} does.

Also, to improve our version of Hysteresis to be executed in parallel and to get better performance results for comparison with OpenCV, once that our implementation use a serial implementation of Hysteresis.

Another interesting work is to map theses DSLs and their IRs, as well as their target architectures to explore performance relations between intermediate representations and architectures.

Bibliography

[Appel, 1960] Appel, A. W. (1960). *Compiling with Continuations*. Cambridge University Press.

[Arbelaez et al., 2007] Arbelaez, P., Fowlkes, C., and Martin, D. (2007). The berkeley segmentation dataset and benchmark 2007. `http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/`. Acessado em 04/05/2017.

[Canny, 1986] Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence VOL. PAMI-8, No. 6, November 1986*, 8(6):210–224.

[Köster et al., 2014] Köster, M., Leißa, R., Hack, S., Membarth, R., and Slusallek, P. (2014). Code refinement of stencil codes. *In Parallel Processing Letters (PPL)*, 24(3):1–16.

[Leißa et al., 2015] Leißa, R., Köster, M., and Hack, S. (2015). A graph-based higher-order intermediate representation (2nd place: Artifact evaluation for cgo/ppopp'15). In *In Proceedings of 2015 International Symposium on Code Generation and Optimization (CGO)*, pages 202–212, San Francisco, CA, USA.

[Lourenço, 2011] Lourenço, L. H. A. (2011). Paralelização do detector de bordas canny para a biblioteca itk utilizando cuda. Master's thesis, Pós-Graduação em Informática - Universidade Federal do Paraná.

[Luo and Duraiswami, 2008] Luo, Y. and Duraiswami, R. (2008). Canny edge detection on nvidia cuda. In *Communications of the ACM*, pages 1–8, Pattern Recognition Workshop, IEEE Computer Society, Los Alamitos, CA, USA.

[Martin et al., 2001] Martin, D., Fowlkes, C., Tal, D., and Malik, J. (2001). A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. 2:416–423.

[Membarth et al., 2016] Membarth, R., Reiche, O., Hannig, F., Teich, J., Körner, M., and Eckert, W. (2016). Hipacc: A domain-specific language and compiler for image processing. *In Transactions on Parallel and Distributed Systems (TPDS)*, 27(1):210–224.

[Membarth et al., 2014] Membarth, R., Slusallek, P., Köster, M., Leißa, R., and Hack, S. (2014). Target-specific refinement of multigrid codes. In *In Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 52–57, New Orleans, LA, USA.

[Pulli et al., 2012] Pulli, K., Baksheev, A., Kornyakov, K., and Eruhimov, V. (June 2012). Real-time computer vision with opencv. *Communications of the ACM*, 55(6).

[Ragan-Kelley et al., 2013] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Programming Language Design and Implementation (PLDI)*.

[Schmitt et al., 2014] Schmitt, C., Kuckuk, S., Hannig, F., Köstler, H., and Teich, J. (2014). Exaslang: A domain-specific language for highly scalable multigrid solvers. In *In Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51, Izmir - Turkey.

[Tompson and Schlachter, 2012] Tompson, J. and Schlachter, K. (2012). An introduction to the opencl programming model. *In Transactions on Parallel and Distributed Systems (TPDS)*.