NTNU – Trondheim
Norwegian University of
Science and Technology

Performance Comparison of GPU, DSP and FPGA implementations of image processing and computer vision algorithms in embedded systems

Egil Fykse

Master of Science in Electronics
Submission date: June 2013
Supervisor: Bjørn B. Larsen, IET
Co-supervisor: Kristin Paulsen, Kongsberg

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Project description

Graphic Processing Units (GPUs) are immensely powerful processors outperforming Central Processing Units (CPUs) in a variety of applications on supercomputers and PCs. GPUs are now being manufactured for portable devices, such as mobile phones and tablets, in order to meet the increasing demand for performance in this segment.

In embedded systems, Digital Signal Processors (DSPs) and Field-Programmable Gate Arrays (FPGAs) have been the leading processor technologies. The architecture of the GPU is highly parallel and tailored to efficiently construct images of 3D models. It is therefore conceivable that GPUs are suitable for implementations of image processing and computer vision algorithms in embedded systems.

The aim of the project is to compare the performance of the GPU, DSP and FPGA implementations of known algorithms in embedded systems. Algorithms such as normalized cross correlation and Finite Impulse Response (FIR) filters are especially interesting. An evaluation of total project cost should be made where considerations about development time, component cost and power consumptions are included.

Abstract

The objective of this thesis is to compare the suitability of FPGAs, GPUs and DSPs for digital image processing applications. Normalized cross-correlation is used as a benchmark, because this algorithm includes convolution, a common operation in image processing and elsewhere. Normalized cross-correlation is a template matching algorithm that is used to locate predefined objects in a scene image. Because the throughput of DSPs is low for efficient calculation of normalized cross-correlation, the focus is on FPGAs and GPUs. An efficient FPGA implementation of direct normalized cross-correlation is created and compared against a GPU implementation from the OpenCV library. Performance, cost, development time and power consumption are evaluated for the two platforms. The performance of the GPU implementation is slightly better than the FPGA implementation, and less time is spent developing a working solution. However, the power consumption of the GPU is higher. Both solutions are viable, so the most suitable platform will depend on the specific project requirements for image size, throughput, latency, power consumption, cost and development time.

Sammendrag

Målet med denne oppgaven er å sammenligne egnetheten av FPGA, GPU og DSP for digital bildebehandling. Oppgaven ser på normalisert kryss-korrelasjon, fordi denne algoritmen inkluderer konvolusjon, en mye brukt operasjon i bildebehandling og ellers. Normalisert krysskorrelasjon er en template matching-algoritme som brukes for å lokalisere forhåndsdefinerte objekter i et bilde. Fordi dataraten i DSP er noe lav for effektiv beregning av normalisert kryss-korrelasjon, fokuserer oppgaven på FPGA og GPU. En effektiv FPGA-implementasjon av direkte normalisert kryss-korrelasjon utvikles og sammenlignes mot en GPU-implementasjon fra OpenCV-biblioteket. Ytelse, kostnader, utviklingstid og strømforbruk vurderes for de to plattformene. Ytelsen til GPU-implementasjon er litt bedre enn FPGA-implementasjon, og mindre tid er brukt for å utvikle løsningen. Imidlertid er strømforbruket til GPU høyere. Begge løsninger er konkurransedyktige og den mest passende plattformen vil avhenge av de spesifikke kravene i prosjektet til ytelse, strømforbruk, kostnader og utviklingstid.

Preface

This master's thesis in electrical engineering is written at the Institute of Electronics and Telecommunications at the Norwegian University of Science and Technology for Kongsberg Gruppen. Bjørn B. Larsen at NTNU and Kristin Paulsen at Kongsberg Gruppen have supervised the work. I wish to thank them for their help and guidance during my work with this thesis.

Egil Fykse
Trondheim, June 2013

Contents

List of Figures

List of Tables

Abbreviations

CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DFT	**D**iscrete **F**ourier **T**ransform
DSP	**D**igital **S**ignal **P**rocessor
FFT	**F**ast **F**ourier **T**ransform
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
FLOPS	**FL**oating-point **O**perations **P**er **S**econd
GPU	**G**raphics **P**rocessing **U**nit
IP	**I**ntellectual **P**roperty
LSB	**L**east **S**ignificant **B**it
LUT	**L**ook**U**p **T**able
MACS	**M**ultiply-**AC**cumulates per **S**econd
MSB	**M**ost **S**ignificant **B**it
NCC	**N**ormalized **C**ross-**C**orrelation
RAM	**R**andom-**A**ccess **M**emory
SAD	**S**um of **A**bsolute **D**ifferences
SOC	**S**ystem **O**n **C**hip

Chapter 1

Introduction

The purpose of this report will be to examine trade-offs and results in the implementation of template matching by normalized cross-correlation (NCC) in embedded, real-time systems on FPGAs and GPUs. Template matching is a method used in computer vision to scan for the presence of predefined objects in an image.

1.1 Problem definition

As the phrase suggests, template matching involves comparing a template, model, or representation of an object against a scene image to see if it is present, and if so, where in the image it is located. Template matching can be performed in many different ways, using different estimators and similarity measures. This report will focus on the normalized cross-correlation approach. Also called the normalized correlation coefficient or Pearson product-moment correlation coefficient, this is a similarity measure that is invariant to scaling and offset in the input images [1]. This is an important feature, as will be further explained in chapter 2.

In this approach, the template is simply an image of the object we are looking for. Mathematically, the problem to be solved can be stated as finding the maximum of

$$R(x,y) = \frac{\sum_{u,v}(I(u,v) - \bar{I}_{x,y})(T(x+u,y+v) - \bar{T})}{\sqrt{\sum_{u,v}(I(u,v) - \bar{I}_{x,y})^2 \sum_{u,v}(T(x+u,y+v) - \bar{T})^2}} \tag{1.1}$$

where I is the scene image, T is the template to be searched for, $\bar{I}$ and $\bar{T}$ are the means of I and T, (u,v) are the scene image coordinates and (x,y) are the

template coordinates. The rationale behind this expression will be explained in chapter 2.

The implementation of normalized cross-correlation will be evaluated across the different platforms with respect to performance, development effort, cost and power consumption.

1.2 Revised project description

The performance of this algorithm will be explored on FPGAs and GPUs. A comparison with DSP is also a part of the original problem description. This was meant as a lower performance benchmark against which to compare the two other platforms. The DSP intended for use in this project (Analog Devices ADSP-21489) has a peak performance of 2.7 GLOPS (billions of FLoating-point Operations Per Second) [2], compared to 276 GMACS (billions of Multiply-ACcumulates per Second) [3] for the FPGA used and 270 GLOPS [4] for the GPU. In addition, the DSP is mainly targeted towards audio applications, so there are no frameworks or support for doing image processing. These factors combined means that a high effort is needed to create an implementation for this platform, and the result is likely to be under-performing. For this reason and in agreement with Kongsberg Gruppen, the DSP will not be included in the comparison.

1.3 Motivation

Applications for template matching include object identification, vehicle number plate recognition and face recognition. Searching for pre-defined objects of this type is just one application for normalized cross-correlation. Other uses include image registration, motion tracking and medical imaging. An efficient implementation of normalized cross-correlation may contain an efficient implementation of correlation or convolution. This has even wider applications, such as for example image filtering and edge detection. It may also be used in to solve differential equations, which has wide applications.

Normalized cross-correlation is a seemingly straightforward solution to the problem. However, this method presents challenges in a real-time system. The amount of time needed to compute the NCC is predictable, but a huge number of operations are required in a naive implementation [5]. The need for an efficient implementation is necessitated even more by the inherent variability of many real-life image scenarios. To search for a particular object, the image not only has to be scanned once, but many times for different possible sizes and orientations of the object in question.

GPUs and FPGAs are well suited for image processing, because of the inherent parallelism in these problems. FPGAs are a mature technology. GPUs for embedded systems are on the rise because of the demand for high-quality graphics in cell phones and tablets. It is therefore interesting to explore which technology is most suited for new projects.

1.4 Background

Some papers about normalized cross-correlation on GPUs and FPGAs already exist. Details about some existing approaches and results will be given in chapter 2. Some of these papers also give a comparison between GPU and FPGA implementations. One example is [6], in which the GPU implementation has the highest performance. However, as new platforms and technology becomes available, implementation comparisons have to be revisited in order to see if different technologies still compare in the same way. In this report, the implementation details of the algorithms will also be covered in detail, as many papers omit crucial details necessary for re-implementation and evaluation. This report will aim to give a comparison of normalized cross-correlation performance on two concrete platforms. Additionally, the source code of the implemented solution will be available, along with a description of key design choices and implementation details.

1.5 Scope and limitations

In order to achieve good results in the time allocated, some simplifications and pre-built primitives are necessary. This will allow us to focus on achieving a good comparison between the two platforms.

For the FPGA implementation, well-known and much-used primitives will not be re-implemented. This includes functions such as division and square root, for which IP is available from Xilinx at no cost. The rest of the implementation will be designed and implemented manually using VHDL.

Since this is a thesis with a focus on design of digital circuits, the focus of the work will be on the FPGA implementation. For the GPU, a free, open source, optimized implementation of NCC is available through the OpenCV project (for details, see section 2.6.2). This will be used for comparison against the FPGA implementation. Thus, the GPU part will not be implemented from scratch.

Only monochromatic images will be considered for the scene and template images. This simplifies the implementation and allows a more efficient computation, and is a common approach in template matching [1].

1.6 Report structure

This report is organized as follows. This chapter gives an overview of the goals and limitations of the work. Chapter 2 presents theory relevant for efficient computation of the normalized cross-correlation. Details of the methods used and implementation architecture are given in chapter 3. The results and relevant interpretation are given in chapter 4. Finally, chapter 5 presents a discussion and conclusion.

Chapter 2

Theory

This chapter will outline the theory behind the use of normalized cross-correlation for solving the template matching problem, efficient ways of solving it and how these methods may be implemented on FPGA or GPU. Necessary theory of FPGA and GPU architecture will also be presented. Before explaining the significance of normalized cross-correlation, we will present the basics of image filtering, convolution and correlation, as these topics are a prerequisite.

2.1 Image filtering

The idea of spatial image filtering is to have a mask or kernel of a certain extent that applies a desired operation to the image pixels under the kernel. By moving the kernel around the image, so that all pixels are visited by the center of the kernel, the image is filtered. The operation applied to the underlying image may be either linear or non-linear. We will focus only on linear filtering. This implies that the filter kernel will be a matrix of coefficients that will be multiplied with the corresponding underlying image pixels [7]. The sum of all products at each location will yield the pixel values of the filtered image.

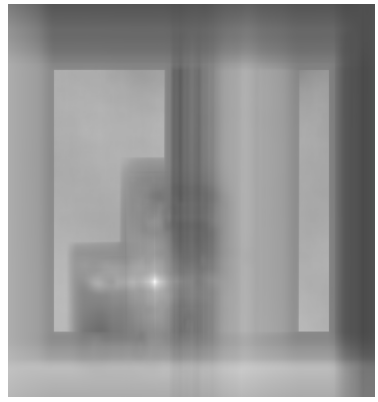
When using symmetric filter kernels, the orientation of the kernel is not important. However, if the filter kernel is non-symmetric, orientation is critical. This is the center of the distinction between convolution and correlation. Correlation is the process described in the previous paragraph. When doing convolution, the same procedure is followed, except that the filter kernel is rotated 180° [7]. It is important to be aware of this distinction, as correlation is the algorithm of interest in the following chapters. If a library function that only supports convolution has to be used, then the kernel (or equivalently, the image) must be rotated 180° before processing.

Another issue that arises in image filtering is boundary conditions. For some image pixels near the border of the image, parts of the kernel will be outside the image boundaries. There are different ways to handle this. Two common methods are zero padding and avoiding boundaries. In zero padding, the center of the kernel is moved over all pixels of the image. When pixels outside the image are needed for the calculation, they are assumed to have zero value. Avoiding boundaries is, as the name suggests, simply a matter of not moving the kernel over the image pixels that are so close to the edge that part of the template would be outside the image. Which of these (or other) approaches is most suitable depends on the application.

2.2 Template matching

Image filtering is the basis of the template matching algorithm mentioned in chapter 1. This algorithm takes as input a scene image, which is the image that will be searched in, and a template image, which is the image that will be searched for. When the template is moved across the image like a filter kernel, the output will have strong responses in areas where the input image has high correlation with the template, indicating high similarity. This cross-correlation may be written as $C = \sum_{u,v} I(u,v)T(x+u, y+v)$, where I is the scene image matrix and T is the template image matrix. A visual explanation of the concept is shown in figure 2.1.

(A) Scene image with template image superimposed.



(B) Result of template matching.

FIGURE 2.1: A visual explanation of template matching. Image A indicates how the template of the door handle is scanned across the scene image. Image B shows the result of template matching. Note the bright white spot at the location of the door handle in the scene image.

Another important step of the template matching method is normalization. Simply correlating the scene image with the template image is not enough to achieve working template matching under real-world conditions. Real images of the same object under different conditions are not pixel-wise identical. One source of variability is lightning conditions. In very bright parts of the image, the output will have a strong response even if the object we are searching for is not present. On the other hand, if the object is visible in the image, but appears dark for example due to poor lightning, it will generate a weak response in the output and may not be detected. To alleviate this problem, normalization is applied after correlation. This is called normalized cross-correlation.

Instead of simply correlating the input image with the template, the mean of the part of the image that is under the template is first subtracted from the image. This subtraction may be written as $I(u,v) - \bar{I}_{x,y}$, where $\bar{I}_{x,y}$ is the mean of the part of the image that is under the template at the current position. In the same way, the mean of the template is also subtracted from the template, written as $T(x+u, y+v) - \bar{T}$. The two are then correlated and the result is divided by the standard deviation of the image and the template, yielding the formula shown in (2.1) [8].

$$R(x,y) = \frac{\sum_{u,v}(I(u,v) - \bar{I}_{x,y})(T(x+u,y+v) - \bar{T})}{\sqrt{\sum_{u,v}(I(u,v) - \bar{I}_{x,y})^2 \cdot \sum_{u,v}(T(x+u,y+v) - \bar{T})^2}} \tag{2.1}$$

2.2.1 Strengths and weaknesses

Even template matching using normalized cross-correlation has several shortcomings from a computer vision viewpoint. One problem is the unreliability of the algorithm. Because it relies on a pixel-by-pixel comparison, the algorithm is sensitive to variations both in the object itself and in the imaging conditions [5].

Despite these limitations, it is a widely used algorithm with some advantages [9]. The normalization step helps somewhat effectively in overcoming illumination changes. It is also relatively simple to understand and implement, at least in a straightforward way.

2.3 NCC implementations

Because a high number of templates may need to be matched against each scene image, an efficient algorithm and implementation is required. Many systems may also have real-time requirements, and predictable timing may be necessary. The following sections will address the problem of efficient implementations.

A minor issue that may arise is that if the scene image has zero variance under the template at any position, the denominator of (2.1) will be zero at that position. This must be addressed to avoid divide-by-zero situations, which may have unpredictable results. This also means that the normalized cross-correlation is undefined at these positions. The MATLAB implementation of normalized cross-correlation assigns zero output in that case [10].

2.3.1 Straightforward implementation

Calculation of the normalized cross-correlation may be calculated directly by implementing (2.1) in a straightforward manner. $R(x, y)$ is then calculated for one pixel at a time. This includes finding the correlation between the scene image and the template at each (x, y)-pair, as well as the sum and the sum of the squared scene image values that lie directly under the template at that position.

Throughout this document, let X and Y be the height and width of the scene image, respectively. In the same way, let M and N be the height and width of the template image. For each position of the template in the $X \times Y$ picture, the correlation across the $M \times N$ template must be evaluated. Therefore, a naive implementation will require on the order of $M \times N \times X \times Y$ multiplications and additions. The problem is made worse by the fact that template matching requires the size and orientation of the object to be the same in the image as in the template. In order to have robust detection, each image must be cross-correlated with many templates of varying size and with varying orientation of the object. This can dramatically increase the number of operations needed. This implementation must iterate over all scene image pixels, and for each pixel it must iterate over a number of pixels corresponding to the number of pixels in the template. Let the scene image size be $X \times Y$ and the template size be $M \times N$, as defined in the previous section. Then the total number of calculations (in big-O notation) will be $O(X \times Y \times M \times N)$.

2.3.2 DFT-based correlation

This section will outline how the discrete Fourier transform may be used in efficient calculation of normalized cross-correlation.

As shown in appendix B, the numerator may be written as a difference between two terms. The first term is the correlation between the template and the part of the scene image that is under the template. Direct calculation of this term for each pixel is expensive. Instead of using the direct method on each pixel, a DFT-based approach may be used to calculate the correlation values for the entire scene image, as detailed in the following sections.

The DFT is a linear transform from the spatial domain to the frequency domain. Let $f(x, y)$ be an image of size $X \times Y$. Then the definition of the two-dimensional discrete Fourier transform is

$$F(q, r) = \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} f(x, y) e^{-j2\pi(qx/X + ry/Y)} \tag{2.2}$$

The correlation theorem states that correlation in the spatial domain is equivalent to multiplication in the frequency domain [7]. Let I be the scene image and T be the template image. The cross-correlation C may then be found as

$$C = I \star T = \mathfrak{F}^{-1} \left\{ \mathfrak{F} \left\{ I \right\} \mathfrak{F}^{*} \left\{ T \right\} \right\} \tag{2.3}$$

The superscript star indicates the complex conjugate of the Fourier transform. Complex conjugation in the frequency domain is equivalent to reflection in the spatial domain. This means that instead of using the complex conjugate, the template may be flipped before taking the Fourier transform.

The one-dimensional discrete Fourier transform may be calculated in $O(n \log n)$ time using a Fast Fourier Transform algorithm (FFT). This is what makes DFT-based correlation worthwhile compared to direct correlation in the spatial domain.

Two-dimensional DFT

As noted in [11], the DFT of a two-dimensional image may be computed by the use of one-dimensional DFTs. This is convenient if a one-dimensional DFT function is easily available, but not a two-dimensional. This is the case in FPGAs.

To compute the two-dimensional DFT, take the one-dimensional DFT off all the columns in the image matrix first. Then, apply a one-dimensional DFT to the rows of the result from the previous step. The order of operations (i.e. with respect to rows and columns) may be interchanged. This may be written as

$$\begin{aligned} H &= \text{FFT-on-index-1}(\text{FFT-on-index-2}[h]) \\ &= \text{FFT-on-index-2}(\text{FFT-on-index-1}[h]) \end{aligned} \tag{2.4}$$

as noted in [11].

This method requires many data accesses in order to copy the data to and from memory. If a multidimensional DFT algorithm is available, it should be used

[11]. However, for implementation on an FPGA the method is still viable. A major advantage in this case is that optimized one-dimensional FFT IPs are readily available. In addition, both the row and column operations may be performed in parallel on an FPGA.

The number of operations necessary is as follows: $O(X \log X \times Y \log Y)$ operations to transform the scene image from the spatial domain to the frequency domain. The same number of operations is needed to transform the template, even though it is smaller than the scene image. This is because the template must be padded with zeroes before taking the transform, so that the frequency domain representation of the scene image and template image will be of the same size. This is crucial in the next step, where the frequency matrices will be multiplied together element by element. This requires $O(X \times Y)$ operations. Finally, the inverse Fourier transform takes the same number of operations as the forward transform. This means that in total, $O(X \log X \times Y \log Y)$ operations are needed. For large templates, this will be much faster than the direct approach, which requires $O(X \times Y \times M \times N)$ operations.

FFT IP

FFT Intellectual Property (IP) makes the transformation to and from the frequency domain more convenient. An example of an FFT IP is the Xilinx LogiCORE IP Fast Fourier Transform v8.0 [12]. It implements the Cooley-Tukey FFT algorithm for transform sizes $N = 2^m$ where m is an integer $m \in [3, 16]$ and word sizes in the range of $8 - 34$ bit. It supports run-time configurable transform length, radix-2 and radix-4 FFT architectures and scaled and unscaled integer operation. These and additional options are available in the LogiCore IP generator.

The FFT core operates in stages called butterflies. For a thorough treatment of FFT theory, see [13]. The numbers returned after each butterfly step may be larger than the input numbers. This means that either more bits must be used to store the output than input or a scaling factor must be applied after each step. In practice, this is implemented as a per-frame configurable scaling schedule that must be supplied to the core if it operates in the scaled mode. The output after each butterfly may be shifted by 0, 1, 2 or 3 bits. It is vital to avoid overflow, or else the values will wrap around and the result will be useless. The core has an optional overflow flag that warns the user if this happens. Xilinx describes a conservative scaling schedule that may be employed in order to completely avoid overflow [12].

In order to efficiently explore the design space with regard to scaling, bit lengths and other options, Xilinx offers a bit-accurate C model of the IP core [14]. This model also has a MEX interface for MATLAB. Once compiled, the model can be used as part of MATLAB scripts to model the system.

2.3.3 Fast Normalized Cross-Correlation

J. P. Lewis describes an efficient way of implementing normalized cross-correlation in [8]. The paper notes that while FFT-based correlation can be used to speed up the cross-correlation step, there are still some parts of (2.1) that are computationally expensive if done in a naive way. For example, the expression $I(u,v) - \bar{I}_{u,v}$ represents the sum of the image under the template, with the mean of the part of the image that is under the template subtracted. In order to calculate this expression efficiently, tables of running sums are proposed. Calculating the expression directly would require iterating over all the pixels under the template for each value of x and y. Instead, the running sum of the image may be pre-calculated as $S(x,y) = I(x,y) + S(x-1) + S(y-1) - S(x-1, y-1)$. Then, the sum of the image under an $M \times N$ sized template with the upper left corner positioned at (x,y) may be found as

$$\begin{aligned}\sum_{u=x}^{x+M-1} \sum_{v=y}^{y+N-1} I(u,v) = S(x-1,y-1) - S(x+M-1,y-1) \\ - S(x-1,y+N-1) + S(x+M-1,y+N-1)\end{aligned} \tag{2.5}$$

The exact same logic applies when calculating $\sum_{u,v}(I(u,v) - \bar{I}_{u,v})^2 = \sum_{u,v}(I(u,v)^2 - 2I(u,v)\bar{I}_{u,v} + \bar{I}^2_{u,v})$.

Using this algorithm reduces the running time for finding the denominator of 2.1 from $O(X \times Y \times M \times N)$ to $O(X \times Y)$ at the cost of an $O(X \times Y)$ increase in memory for storing the running sum tables.

2.4 Exploiting parallelism

One of the advantages of platforms like FPGAs and GPUs is the architectural parallelism in these devices. On the other hand, they typically have lower clock frequencies than conventional CPUs. This means that in order to achieve a speed-up, the parallelism of the device must be exploited by identifying and taking advantage of parallelism in the problem to be solved.

Assume that a fraction P of a problem can be parallelized, and that there are N parallel data paths available. Then Amdahl's law states that the maximum achievable speed-up achievable through parallelization is

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.6}$$

This law assumes that nothing changes other than breaking the parallelizable section of the problem up into parts that are executed in parallel. As more parallelism is added (N increases), the return gained by increasing N even more decreases. Another important point is that P should be as large as possible in order to achieve the best effect. The key takeaway is that exposing and exploiting parallelism in the problem is essential to achieve a speed-up on these parallel platforms.

It is important to note that Amdahl's law gives a maximum achievable speed-up, not the speed-up that will in fact be obtained. The actual speed-up may be limited by for example communication requirements between the problem instances.

2.5 FPGA

FPGAs (Field-Programmable Gate Arrays) are integrated circuits that may be electronically programmed (in the field) to execute any type of functionality [15]. They typically consist of programmable logic cells and interconnects. The exact structure varies across FPGA vendors and families. In this section, the focus will be on the architecture of the Xilinx 7 series FPGAs, and especially on the Zynq Z-7020, which is the synthesis target in this project. Nevertheless, the principles should hold for most modern FPGAs.

Xilinx 7 series FPGAs consists of a two-dimensional structure of different elements that may be used to implement desired functionality. This includes configurable logic blocks (CLBs), block RAM, DSP slices and a switching matrix connecting the various elements together.

The CLBs are where the logic functionality of the FPGA fabric is selected. A CLB typically contains look-up tables (LUTs) that can implement arbitrary binary functions, as well as registers, multiplexers and glue logic. The CLBs and the other components are connected together by a matrix of wires and switchboxes, allowing flexible routing of signals between the components. The connectivity of the switching matrix as well as the function of the CLBs is configured by loading a bitstream into the FPGA after power-up.

2.5.1 Block RAM

Block RAM is dedicated memory used for data storage in the FPGA. Xilinx 7 series FPGAS have between 30 and 1,880 blocks of 36 Kb block RAMs (the Zynq Z-7020 has 140) [3]. Each of these may also be used as two independent 18 Kb block RAMs. Each block RAM can have two independent read/write ports of variable size (including 2 Kb $\times$ 8 bit used in this project), and even

differing aspect ratios. The block RAM is fully synchronous and can only be read or written once per port per clock cycle.

2.5.2 DSP Slices

The Zynq series SoCs feature between 80 and 900 DSP slices (220 in the Zynq Z-7020). Each slice has a 25 bit $\times$ 18 bit multiplier. The multiplier (which can be bypassed) feeds its result into a 48-bit accumulator, resulting in a Multiply-Accumulator circuit.

2.5.3 Arithmetic

As can be seen from (2.1), calculating the normalized cross-correlation involves calculation of additions, subtractions, multiplications, division and square roots. Multiplications are implemented automatically by the synthesis tool, either by using DSP slices or FPGA slice logic. Adders and subtractors are implemented automatically in slice logic by default (leveraging dedicated carry logic in the FPGA) [16]. Dividers are implemented using slice logic by default. This results in extremely long timing paths as no pipelining is employed. Square root is not built into the compiler. Thus, a different approach as detailed below is needed for division and square root operations.

Xilinx provides an IP Core for integer division [17]. The divider core features two modes of operation: Radix2 and High Radix. Xilinx recommends the use of the High Radix algorithm for input operands of width larger than 16 bits. In High Radix mode, the latency (in clock cycles) of the calculation may be selected when the divider core is generated. There is a trade-off between latency and resource usage.

The extended CORDIC (COordinate Rotation DIgital Computer) algorithm is a very attractive way of implementing the square root functions in FPGAs, because the only operations required are shifting, adding, subtracting and table look-ups [18]. Xilinx provides an IP Core that implements this algorithm [19], so it does not need to be implemented from scratch. The input number can be an unsigned integer width a maximum width of 48 bits. The pipelining mode can be selected when generating the IP Core using the Xilinx Core Generator. There are three choices: no pipelining, optimal pipelining (using no additional LUTs) or maximum (pipelining after every step of the algorithm).

2.5.4 Row buffering

The motivation for row buffering is to both make effective use of the memory available on the FPGA. If the image is stored off-chip, row buffering can

minimize off-chip memory accesses. Consider a search template being moved sequentially over an input image. As the template moves along, only the number of pixels in a single column of the template must be loaded from the input image. The rest have already been referenced. If they are cached in the row buffer, the amount of memory accesses are kept to a minimum. When the image enters the FPGA on a row-by-row basis, row buffering makes the most effective use of on-chip memory. The row buffer will store all the rows of the input image that the template is currently being moved over, as shown in figure 2.2.

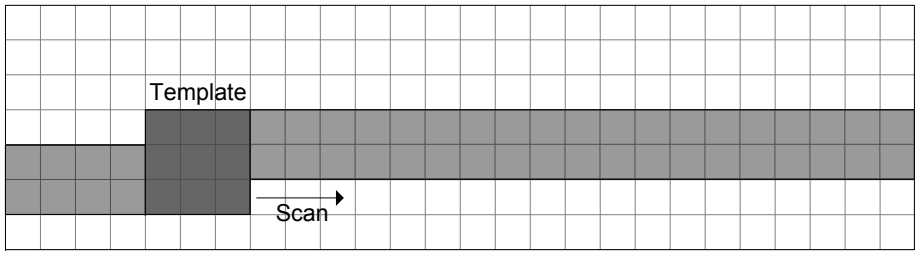


FIGURE 2.2: Illustration of row buffering. The template (dark grey) is moved across the image. The pixels (light grey) are stored in the row buffer. [20]

Ideally, the row buffer would be implemented as a FIFO buffer. This means that the row buffer will only store the pixels needed, and as each new pixel is added, another is removed from the buffer in a FIFO manner. A FIFO implementation has the advantage that the pixels actually needed for the calculations (that is, the pixels directly under the template) are in the same location in the buffer. This means no multiplexors are needed, as each pixel position will always be on the same index in the buffer memory. Xilinx 7 series FPGAs have built-support and logic for block RAM FIFOs. Unfortunately, the contents in the middle of the FIFO buffer are not available. Only the output word at the end of the buffer may be read. This makes the built in FIFO unsuitable as a row buffer, as the contents in the FIFO buffer are needed to calculate the cross-correlation.

A row buffer may also be implemented with shift registers, but this has a major disadvantage. Such an implementation would use many flip-flops and therefore a large number of the CLBs available on the FPGA. Block RAM is more suited for storing image data in such large quantities [20].

2.5.5 An FPGA solution architecture

Wang and Wang describe a way of computing the straightforward normalized cross-correlation on an FPGA [21]. The paper is rather practical and describes two possible architectures. Performance and area results for each architecture are given. A major point in their architecture is the parallel computation of

several multiplications and additions, when computing the correlation between scene image and template. They also propose a memory architecture with row buffering and multiplexers in order to achieve this parallelization of arithmetic operations. For a scene image size of 512×512 and a template image of size 80×80, they achieve a clock frequency of 70 MHz and a latency of 224 ms.

2.5.6 Simulation

Functional simulation is straightforward using VHDL simulation software. However, if IP cores are used in the design to be simulated, the necessary libraries must be available to the simulation tool. They are available from the FPGA vendor.

2.5.7 Synthesis

Synthesis, followed by mapping, place and route must be done before the circuit described in VHDL can be implemented on an FPGA and the performance of the circuit can be known. Synthesis builds a representation of the state machines in the design and the relationship between them. The logic is optimized to remove redundancy. Mapping replaces the high-level constructs created during synthesis with actual component types available on the target FPGA. This included LUTs, block RAM and DSP units. Place and route tries to place the components and interconnects so that timing constraints can be met.

These steps are done using automated tools, but the tools may have a wide variety of options that will affect the resource usage, performance and power consumption of the final circuit. Additionally, different synthesis tools may not give equal results when processing the same set of VHDL files.

2.6 GPU

GPUs (Graphics Processing Units) are characterized by their parallel structure, with a high number of processors compared to traditional CPUs. Originally used for computer graphics, they can now be used for all types of calculations. The highest speedups are achieved when the problem can be partitioned into smaller, independent blocks that each run the same code, and does minimal conditional branching. This means that it is very well suited for image processing algorithms, where the image may be partitioned into smaller blocks, and each block is processed on one of the many GPU cores.

2.6.1 Architecture

In GPU computing, code is typically executed both on a conventional CPU and on a GPU. Sequential code with many random memory accesses, branches and complicated logic is most efficiently executed on the CPU. Parallel, arithmetic code may see a significant speed-up when executed on the GPU [22].

Efficient development of parallel applications for GPUs is dependent upon a sufficiently high-level programming language. Examples of such languages are CUDA (for NVIDIA GPUs) and OpenCL (general-purpose). These languages helps abstract the complicated nature of the GPU hardware, hiding low-level features. By focusing on exploiting the inherent parallelism of the problem, significant speed-ups can be gained with reasonable effort.

The parallel part of a CUDA program consists of kernels that are executed on the GPU. Many copies of the kernel are executed in parallel as threads, grouped together in thread blocks. Threads within the same block may communicate through shared memory.

2.6.2 The OpenCV library

The Open Source Computer Vision Library (OpenCV) is a library of functions used in computer vision and image processing. It is cross-platform, so that it may be compiled for ARM and executed on the CARMA board. As of the latest version (2.4.5), the necessary makefiles to compile for CARMA are included in the library. It has about 500 algorithms and 5000 functions, many of which have specialized GPU implementations along with their (multi-core) CPU counterparts. This includes functions for template matching and image filtering. The library is written in C++, optimized for speed and is free for commercial applications under a BSD license.

The OpenCV template matching function `matchtemplate()`, when called with the method flag set to `CV_TM_CCOEFF_NORMED`, calculates the normalized cross-correlation R, as given in equation 2.1 [23].

The source code for `matchtemplate()` [24] reveals that the implementation closely follows the one suggested in [8]. The `matchtemplate()` function will determine the method to be used for the matching and the data type used in the image and template (they must be identical). Based on this, it will call the appropriate function (for example `matchTemplate_CCOFF_NORMED_8U` if the method is set to `CV_TM_CCOEFF_NORMED` and the image and template values are 8-bit unsigned integers). The sequence of function calls after this is shown in figure 2.3.

Based on the area of the template, the library will choose between the naive, direct correlation method if the area is below the threshold and the FFT-based correlation method otherwise. Both methods are executed on the GPU. The threshold value for the template area is hardcoded at 250 pixels in the `getTemplateThreshold` function.

Once the scene and template image are cross-correlated, they will be normalized using the `matchTemplatePrepared_CCOFF_NORMED_8U` function, which also utilizes the GPU. First integral tables are prepared, as described in section 2.3.3. Then those the tables are used to do the normalization.

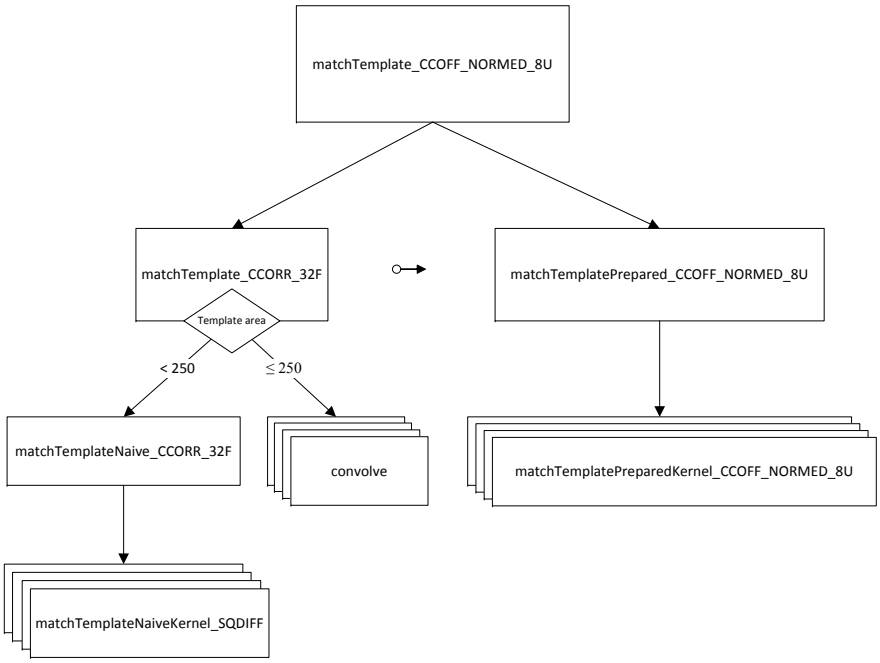


FIGURE 2.3: Data flow of the `matchTemplatePrepared_CCOFF_NORMED_8U` function. The downward arrows show function falls. The horizontal arrow shows flow of data. Parallelism is indicated by several boxes on top of each other.

Chapter 3

Method

This chapter will detail how the solutions to the normalized cross-correlation problems are designed. Specifications, design choices, architecture and test procedures will be discussed.

3.1 Development equipment

The comparisons and results of this project are made with specific hardware in mind. This section will give an overview of the target platforms.

3.1.1 ZedBoard (FPGA)

ZedBoard is an evaluation and prototyping board featuring the Xilinx Zynq Z-7020 All Programmable SoC. Devices in the Zynq series have a dual-core ARM Cortex A9 processor and 28 nm Xilinx programmable logic. The programmable logic in the Z-7020 is equivalent to that of an Artix-7 FPGA, and has 53,200 LUTs and 560 KB block RAM[3]. A picture of the ZedBoard is shown in figure 3.1.

3.1.2 CARMA (GPU)

The CARMA development kit from SECO will be used for GPU software development and testing. The development board features an NVIDIA Tegra 3 ARM Cortex A9 Quad-Core CPU and an NVIDIA Quadro 1000M GPU. The Quadro 1000M GPU has 96 CUDA cores with compute capability 2.1, 2 GB

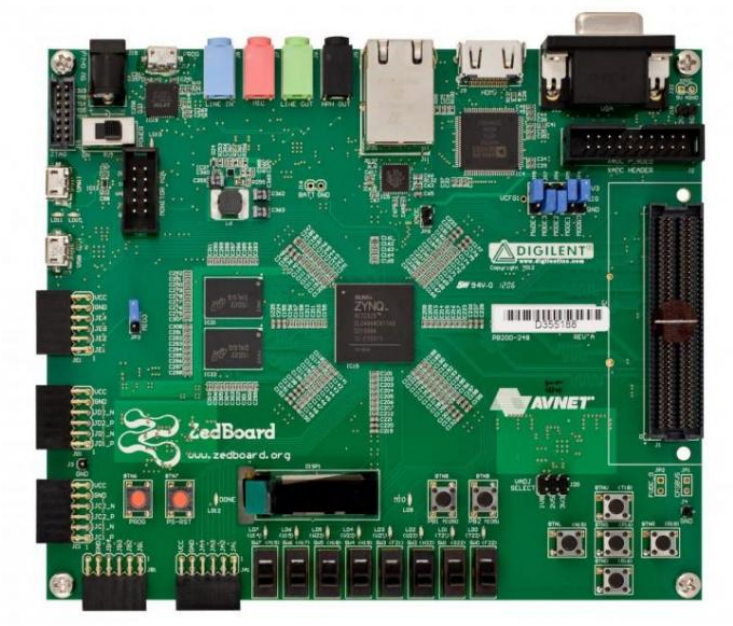


FIGURE 3.1: ZedBoard development kit.

DDR3 memory and a maximum power consumption of 45 W [25]. A picture of the CARMA board is shown in figure 3.2.

The kit comes pre-loaded with a specially built, stripped-down Linux version. One can either connect directly to the card using USB keyboard and mouse and a HDMI-compatible monitor, or operate remotely through Ethernet and SSH. The compiler needed to compile CUDA-enabled programs for ARM cannot run on the CARMA card itself. For details about how to set up the compiler, see appendix A.

3.2 FPGA

First, a method for implementing the straightforward normalized cross-correlation in accordance with (2.1) will be presented.

3.2.1 Specification

The FPGA implementation of direct normalized cross-correlation will have an interface as shown in table 3.1. The `template` and `image` ports are 8-bit wide

FIGURE 3.2: CARMA development kit.

ports, for loading the template and scene image as unsigned integers. The image and template data will be sampled on the rising edge of the `image_clk` signal. In addition to the image clock, a system clock signal must be supplied to the circuit (`clk`). The clock must run at a faster rate than `image_clk`, because many operations must be executed per pixel. Exactly how much faster it has to run depends on the implementation. When `start` is asserted, loading of the template will commence. When the template is finished loading, the scene image will be loaded and normalized cross-correlation processing will start. The circuit will output the normalized cross-correlation data for verification purposes.

TABLE 3.1: Interface to the normalized cross-correlation circuit.

Port name	Direction	Size	Description
image	in	8	Scene image input
template	in	8	Template image input
start	in	1	Starts loading of data and calculation of result
image_clk	in	1	Clock for image and template loading
clk	in	1	System clock used for calculations and control
reset	in	1	Synchronous reset, active high
ready	out	1	Signals that the system is idle and ready to process data
result_ready	out	1	Signals that the output on normxcorr is valid
normxcorr	out	9	The normalized cross-correlation result

3.2.2 Architecture

In order to achieve a speedup from an FPGA implementation, parallelism in the problem must be identified and exploited, as noted in section 2.4. It is not sufficient merely to port a serial software implementation to the FPGA. This is because FPGAs have much lower clock frequencies than traditional CPUs.

For every possible template position, three quantities are time-consuming to compute directly. These are the correlation between the image and the template, the sum of the image under the template and the sum of the squared image under the template. The method with running sums discussed in section 2.3.3 speeds up the last two calculations, but this relies on a trade-off between speed and memory usage. On FPGAs, memory is a scarce resource. Storing an image with 1024×768 pixels would require 1024×768 pixels $\times 8$ bit $= 786$ KB, exceeding the block RAM capacity of the FPGA. The running sum tables would be even larger, as more than 8 bit would be needed per pixel position. Even storing just a band of height N of the running sum tables is impossible.

Parallelization of the straightforward cross-correlation equation can be employed instead. One possible parallelization opportunity (as noted in section 2.5.5) is in the step where the template pixels are multiplied with the scene image pixels under the template and the results added together. Multiple multiplications and additions can be performed per clock cycle. The number of parallel operations possible is limited by the FPGA resources, such as multipliers and slice logic for adders. The same strategy must be employed for squaring the image pixels under the template, doubling the number of required multipliers.

When doing the cross-correlation, data from both the template and the scene image must be available simultaneously, as they must be multiplied together pixel-by-pixel. Consequently, a memory architecture that allows both the template and scene image to be loaded from memory in the same clock cycle is desirable. One such architecture is shown in figure 3.3. Each row of the figure represents one 18 Kb block RAM. N block RAMs are used, so that the template and the image rows under the template can be stored. This means that the height of the template is limited to the number of available block RAMs (280). The first part of each block RAM is used for storing a scene image row, and the last part stores the template. As noted in section 2.5.1, the block RAM in Xilinx 7 series FPGAs support two simultaneous reads from a single block RAM in one clock cycle. Hence, this architecture ensures that for each clock cycle, one column of the scene image under the template can be loaded, as well as one column of the template.

Writing to the scene image block RAM will be designed to wrap around. This means that when the first N rows of the scene image have been written into block RAM, writing continues from the start of the first block RAM, overwriting its

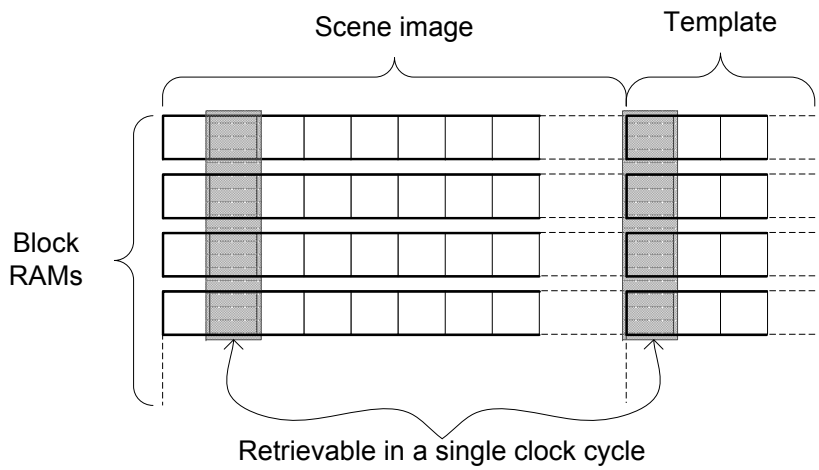


FIGURE 3.3: Diagram of the memory architecture chosen for the implementation. The rows indicate 18 Kb block RAMs. The data indicated in gray can be retrieved in a single clock cycle.

contents. A counter is therefore needed to keep track of which block RAM contains the topmost scene image row.

This memory architecture gives rise to the architecture for the cross-correlation circuit shown in figure 3.4. Before loading the scene image, the template is loaded into the template memory in a pixel-by-pixel manner (this can also be done while loading the scene image, but this is not done here for simplicity). Once the template is loaded, loading of the scene image into block RAM can begin. Let N be the number of rows in the template. As soon as N rows are loaded into the scene image buffer, calculations can begin. Because the block RAM is overwritten in a cyclical manner, multiplexers are needed to shuffle the pixels read from a block RAM column into the correct order. N multiplexers with N byte-wide inputs are needed to be able to shuffle the pixels into the correct order. Once a column from the scene image and a column from the template are loaded in the correct order, the sum, product and sum of the squared image pixels can be calculated. When all columns of the template and underlying scene image have been processed in this way, the results are added together (not shown in the figure) and ready for the normalization step of the calculation. The next scene image pixel can be loaded and processing of that pixel started at this time.

Figure 3.5 summarizes the top-level circuit architecture. The outermost entity (`ncc_top`) contains the normalization logic, and instantiates the entity responsible for cross-correlation, `correlation`. By making the correlation unit a separate module, the correlation strategy can easily be change without affecting the normalization logic, which can be neatly decoupled.

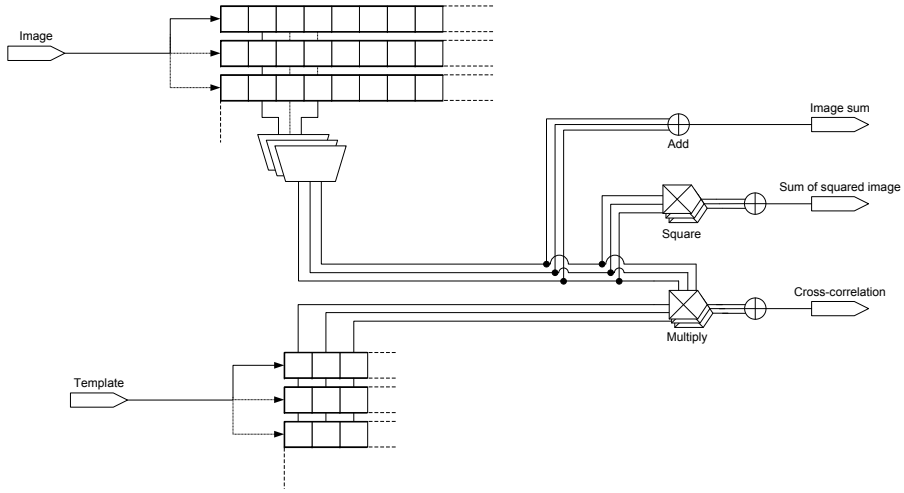


FIGURE 3.4: Architecture of the cross-correlation step. The figure shows how the scene image and template is loaded into block RAM. Multiplexers are used to shuffle the data into correct positions before processing.

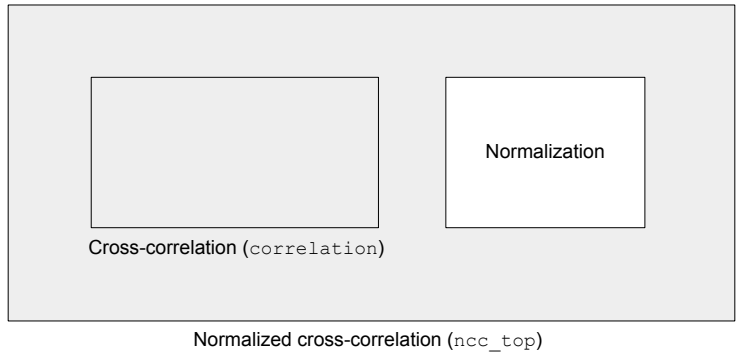


FIGURE 3.5: The top-level architecture of the solution. The grey boxes indicate VHDL entities. The normalization step is part of the `ncc_top` entity.

3.2.3 Implementation

The normalized cross-correlation circuit is implemented in VHDL. The project is managed in Xilinx ISE Design Suite 14.4, using Mentor Graphics Modelsim SE-64 10.2 for simulation and Synopsys Synplify Premier G-2012.09-SP1 for synthesis. The synthesis target is the Xilinx Zynq Z-7020.

Cross-correlation

The cross-correlation module is implemented as a separate entity. The template and scene image are loaded into block RAM as discussed in section 3.2.2. As the template is loaded, the sum of all the pixels in the template and sum of all pixels in the squared template are computed. When the template is finished loading, these values are available on the `template_sum` and `template_squared_sum` outputs.

A state machine is used to control the cross-correlation circuit. After the template has been loaded into block RAM, the scene image starts loading. Once N rows of the scene image are loaded, calculations begin. For each template position, calculation of the output values take M clock cycles, as N pixels are processed in parallel. The calculation is pipelined: one clock cycle is used to retrieve data from memory, one to multiply scene pixels and template pixels as well as squaring image pixels, and one to sum the results for that column. When the calculation is finished, the correlation between the scene image and template at that position, as well as the sum of the scene image pixels and squared scene image pixels under the template are available on `correlation`, `image_sum` and `image_squared_sum`. At the same time, `result_ready` is asserted.

Normalization

Once the results are available from the cross-correlation sub-module, they are normalized to obtain the normalized cross-correlation. The equation for normalized cross-correlation ((2.1)) may be rewritten as

$$R(x,y) = \frac{MN \times \sum (I \times T) - \sum I \sum T}{\sqrt{MN \times \sum_{u,v} I^2 - \left(\sum I\right)^2}\sqrt{MN \times \sum T^2 - \left(\sum T\right)^2 / 2^8}} \tag{3.1}$$

to be better suited for computation on an FPGA, as shown in appendix B. All sums in the equation are over the image template, or the corresponding scene image area directly below the image template. The results of all the sums are available from the cross-correlation sub-module. All that have to be done is multiplying, subtraction, division and square roots.

A state machine controls the normalization procedure as follows. When the cross-correlation at the current position is done, the correlation module will make the results available on the corresponding output ports and assert the `result_ready` signal. This will start calculation of the numerator, as well as the expression under the square root signs in the denominator. The next step is calculating the square roots of the said expressions. Two instances of the LogiCORE IP CORDIC v5.0 from Xilinx (described in section 2.5.3) are used

for the calculation of the square roots. They are configured with the "Optimal" pipelining option, as this is found to give delays that are not in the critical path, while not using excessive amounts of slice logic. The square root calculations finish after 12 clock cycles, as seen in figure 3.6. The results from the square root operations are then multiplied together to form the denominator of (3.1). As noted in the equation, the denominator is divided by 2^8 in order to scale the final result range from $-1.0 \rightarrow 1.0$ to $-256 \rightarrow 255$, avoiding floating-point numbers. The division by 2^8 is implemented by discarding the eight least significant bits of the denominator. After this, the numerator and denominator are ready, and the division is started. Figure 3.7 summarizes the architecture of the normalization circuit.

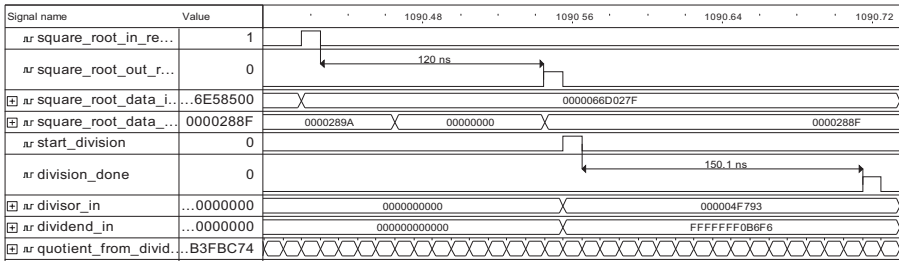


FIGURE 3.6: Timing of the square root and division process. The square root operations take 12 clock cycles, and the division operation takes 15 clock cycles with $t_{clk} = 10$ ns.

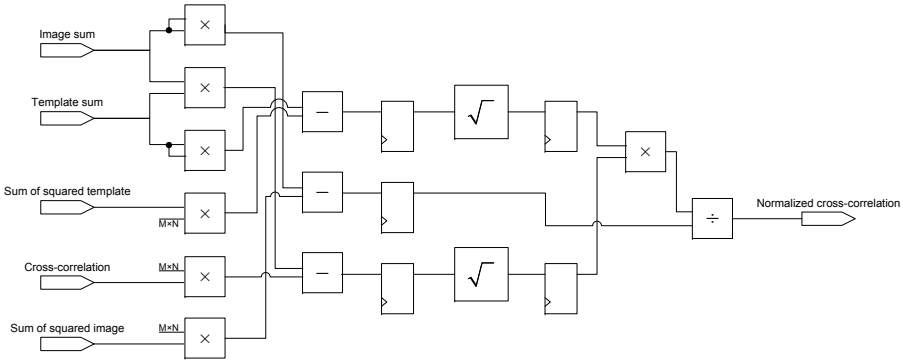


FIGURE 3.7: Architecture of the normalization step. The figure shows how the various outputs from the cross-correlation step are processed to obtain the normalized cross-correlation.

Division is done using the Xilinx LogiCORE IP Divider Generator v4.0 in High Radix mode, as this mode is recommended by Xilinx for input operands larger than 16 bits. When selecting the latency of the divider core, several factors must be considered. A low number of cycles will increase the length of the critical path. If a critical path is found in the divider core, this can be easily

alleviated by increasing the latency. However, one have to make sure that the calculation will be finished in time before the next set of data is available, as pipelining of the calculations is impossible in high radix mode. On the other hand, a lower latency will translate into more slice logic. Through experiments, it was found that a latency of 15 did not cause the critical path to be inside the divider, to this latency was chosen.

According to the manual for the divider core, the result of division by zero is undefined. An output flag called `divide_by_zero` can be used to detect these cases. In this design, we will check if the input denominator is zero before division start. In that case, the numerator will be set to zero and the denominator to one, so that the result of the division will be zero. This is in line with the MATLAB implementation of normalized cross-correlation.

The number of bits used in the intermediate stages of the calculation is an important point when implementing the normalization step. The numbers are represented as `std_logic_vector`s internally, as the integer type is limited to numbers in $[-2^{31}, 2^{31} - 1]$. This is not sufficient for storing the results of the intermediate values. The necessary number of bits depends on the size template used. For instance, the worst case is the sum of a squared template containing only the value 255. This must be taken into account when selecting the number widths.

3.2.4 Test

For functional correctness testing, a testbench is used. The testbench reads in the test data from text files and use this data to stimulate the circuit. The test data consists of 8-bit unsigned integers. They are stored in plain text files with one pixel value listed per line. An efficient way of generating such text files from an existing image is shown below.

```
testImage = imread('in.pgm');
testImage = reshape(testImage.',1,[]);
dlmwrite('image.txt', testImage','newline','pc')
```

During simulation, the calculated normalized cross-correlation at each pixel is saved to file. The circuit is tested for functional correctness by comparing this output against the MATLAB implementation of normalized cross-correlation, `normxcorr2` [10]. An exact comparison is not possible, because of errors introduced by the fixed-point representation. Instead, an error margin of for example 1 % may be employed. If the difference in the output from the circuit and the output from the reference function is within 1 % of the total range of values, the test is passed.

In order not to incur excessive simulation times when verifying the circuit, the test images should be kept to a reasonable size. The test image set used is

taken from a template matching assignment from an image processing course at McGill University [26]. The scene and template images are shown in figure 3.8. The template image will be used to locate the man's nose in the picture using normalized cross-correlation.

(A) Scene image (B) Template image

FIGURE 3.8: The images used for testing the FPGA implementation [26].

To estimate performance, both the clock frequency of the circuit and the total number of clock cycles required for the calculation must be known. The estimated maximum clock frequency of the circuit is reported by the synthesis tool. The number of clock cycles is known through analysis of the circuit architecture, and can be verified through simulation. Then the total time needed for calculating the normalized cross-correlation is $t_{total} = n_{cycles}/f_{clk}$.

3.2.5 Power estimation

The Xilinx XPower Analyzer is used to estimate the power consumption in "Vectorless Estimation" mode. In this mode, actual stimulation vectors are not supplied, instead the tool assigns activity rates to all nodes before calculating power consumption [27].

3.3 DFT-based FPGA implementation

In this section the viability of a DFT-based correlation implementation will be explored. The basis for this exploration will be the Xilinx LogiCore IP Fast Fourier Transform v8.0, as described in section 2.3.2. Before starting the VHDL implementation of the DFT-based correlation, it is a good idea to use the bit-accurate MATLAB model to explore the design space. This is especially

important for the data bit width of the input and output ports, in order to estimate resource and memory usage.

The procedure used for computing the DFT-based correlation is as follows.

1. Map the scene image and template range to the range of the FFT model ($[-1, 1]$).
2. Pad both images so that the resulting images have the same size.
3. Compute the DFT of the columns of the images, and then of the rows of the result.
4. Multiply the Fourier representations together to form the product $\mathfrak{F}\{I\}\,\mathfrak{F}\{T\}$.
5. Compute the inverse DFT of this by first taking the inverse DFT of the columns, and then of the rows of the result.
6. The real part of the result is the DFT-based correlation.

The accuracy of the FPGA-based correlation will be determined as a function of the FFT IP core input and output width. For this purpose, the procedure above will be used, and the results compared using Sum of Absolute Differences (SAD). The DFTs will be MATLABs built-in `fft` function as the gold standard and the bit-accurate Xilinx model as the device under test.

The results of the comparison are shown in figure 3.9. The figure shows unacceptable errors for small data widths. For higher data widths, block RAM will be quickly exhausted because of the wide data widths. As an example, a low data width of 24 is considered. The results after the first DFT stage will require 48 bits per pixel for the scene and template image. As scene image size grows large, this will prohibit storage of the entire image in the FPGA block RAM. This method would have to make use of external memory, or use some way of partitioning the image into blocks that are processed separately, using for example the overlap-save or overlap-add methods [13]. This will not be further considered in this report due to time constraints.

3.4 GPU

Performance measurement of template matching on GPU will be done as a ground for comparison against the FPGA implementation. The main target for comparison is the Quadro 1000M GPU described in section 3.1.2. A brief comparison will also be made against the more powerful, desktop-oriented GeForce GTX 660 Ti. The OpenCV implementation of normalized cross-correlation will be used as the benchmark.

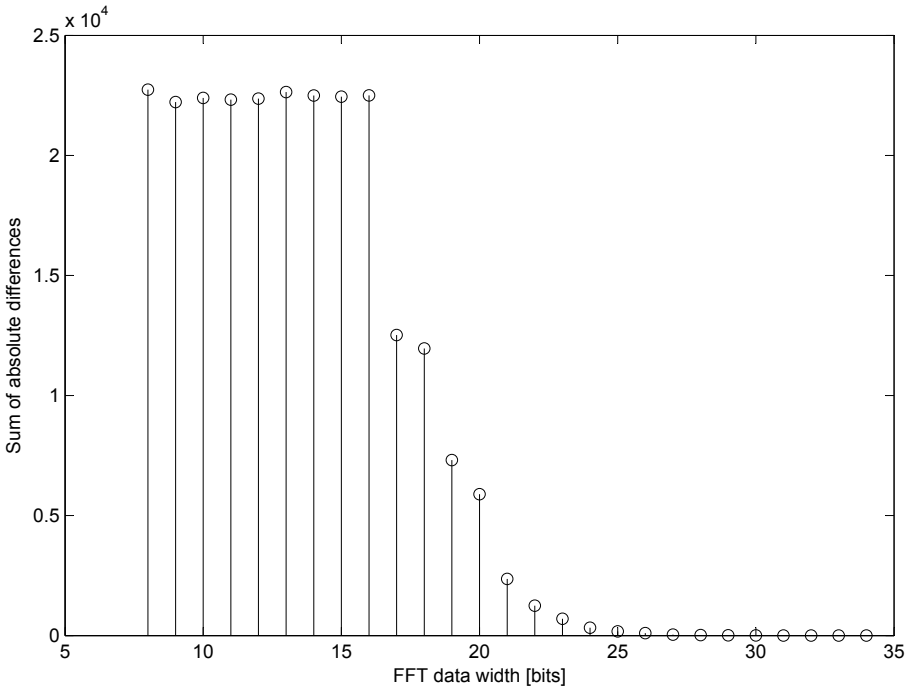


FIGURE 3.9: Sum of absolute differences in the FPGA bit-accurate MATLAB simulation of the DFT-based cross-correlation as a function of DFT data width.

3.4.1 Performance measurement

When OpenCV is compiling and running properly, benchmarking test can be executed. The correctness of the results are confirmed by running the self-test that comes with the OpenCV library (`opencv_test_gpu`). A C++ program that calls the OpenCV functions in question is written and executed to record the time spent calculating the normalized cross-correlation.

The program generates quadratic scene and template images of varying size, calls the OpenCV `matchTemplate` function and records the elapsed time. The scene image side length will be varied in steps of 32. For clarity and to ensure smooth graphs, the template side length will be varied in steps of one.

There are some finer points that one has to be aware of in order to get correct timing results. That is, OpenCV will only initialize the CUDA Runtime API on the first CUDA function call. In addition, OpenCV uses Just In Time-compilation for CUDA code [28]. This means that the first CUDA function call will be very slow (up to several seconds). Therefore, a dummy call must to be

placed to the desired function before the actual performance measurement is done.

For simplicity, random matrices of different sizes may be conveniently prepared through the OpenCV `randn` function. Before the matrices can be populated, they must be created as `cv::Mat` objects. The data type `CV_8U` is used, indicating an unsigned 8-bit integer single-channel matrix.

The simplest way of measuring elapsed time for the function call is to use a CPU timer. However, the recorded time would then include the time spent transferring data to and from the GPU. In order to only record the time actually spent calculating on the GPU, a GPU timer may be used. A CUDA event is then inserted into the CUDA instruction stream before the `mathTemplate` function call, and another one after. The time in milliseconds elapsed between the events according to the GPU clock can then be found using the `cudaEventElapsedTime` function.

Once all the above is set up correctly, one can iterate through all desired combinations of scene image and template size and record the results. On Linux, the procedures detailed in appendix A are used to compile the benchmarking program. On windows, Microsoft Visual C++ 2010 Express is used, together with OpenCV 2.4.5 and CUDA 5.0.

3.4.2 Data analysis

The benchmarking program outputs the time in milliseconds spent calculating the normalized cross-correlation for each scene image and template size. A MATLAB program is created to sort, filter and visualize the results to produce the graphs presented in chapter 4.

3.4.3 Power measurement

Power consumption from the CARMA card is measured by using a TTi EL302 power supply. The supply is configured for 19 V and connected to the CARMA card. Current consumption is measured both in the idle state (only the operating system is running) and during the benchmark test.

Chapter 4

Results

In this chapter, the results obtained for the various platforms will be presented and analyzed.

4.1 FPGA

4.1.1 Cost and development time

Giving an estimate for the development time and effort needed to complete this FPGA implementation is not easy. The time spent during the work with this thesis may not be the same time spent on a similar project by a more experienced developer. Quite some time was spent on getting the development environment running properly.

As more code and IP cores are added to the project, simulation and synthesis time started to take up a significant portion of the total development time. The square root IP core has a precompiled behavioral simulation model that makes for very fast simulation. However, the divider and FFT cores have structural VHDL simulation models. In these models, every internal signal, LUT, DSP48 core and other components are instantiated directly in VHDL. This results in very large models, running into hundred thousands of lines. This slows down the simulation considerably, especially if the simulator intentionally limits the simulation speed. This is the case with Aldec Active-HDL EDU edition. A simulation with a small picture may then take tens of minutes. Mentor Graphics ModelSim SE was available and runs at full speed, so this was used instead for simulating the design when these IPs were included.

In summary, the FPGA development effort was the most time-consuming part of this project. Several weeks were spent coming up with a feasible architecture, implementing it in VHDL, selecting and configuring IP blocks and testing and optimizing the design.

4.1.2 Synthesis

The normalized cross-correlation circuit described in section 3.2 was synthesized with different values for the template size. The scene image size was kept constant at 512×512 pixels. The synthesis results can be seen in table 4.1.

Table 4.1: Synthesis results.

N [pixels]	f [MHz]	LUTs	DSP48s	BRAMs
4	149.0	3046	30	1
8	149.0	3316	38	9
16	149.0	4079	54	17
32	149.0	5289	86	33
64	149.0	8067	150	65
96	145.5	20872	214	97
128	144.5	20031	218	129

After synthesis, the resource requirement for each VHDL entity is reported. From the synthesis report for the 96×96 template circuit, we find the hierarchical resource usage breakdown shown in table 4.2.

Table 4.2: Hierarchical breakdown of resource usage.

Module name	LUTS	Registers	DSP48s	BRAMs
ncc_top	134	272	8	0
correlation	20017	1249	193	96
divider	792	413	13	1
sqrt_1	723	392	0	0
sqrt_2	723	390	0	0

Except for the case where $N = 4$, the number of block RAMs required are $N+1$ as one block RAM is required for the divider core, and N for the correlation, as each row under the template must be stored at all times. The number of DSP48 elements used in correlation is $2N + 1$. For each row in the template, one multiplier is used to find the correlation and one to square the image under the template. In addition, one multiplier is needed to square the template

pixel values as they enter the circuit, before they are added together. Seven multipliers are needed for the normalization process (see figure 3.7). However, if both M and N are powers of two, those multiplications can be implemented by shifting, and only four multipliers are needed. This translates into eight DSP48 slices because of the wide operands (two slices are cascaded to form a larger multiplier).

The limiting factor for the template size is the number of DSP48 slices and block RAMs available. If the number of rows in the template is increased too much, there will not be enough block RAMs to store the template and the scene image under the template. However, for this design, the DSP48 limit will be reached first. When the number of multipliers needed exceed the number of multipliers on the FPGA, the multipliers will be implemented using slice logic instead. This will greatly increase the delay of the circuit, and consume large amounts of area. So to increase the number of rows in the template further, an FPGA with more block RAM and DSP48 slices would be needed.

4.1.3 Performance

The performance of the circuit describes the throughput that is possible to achieve, as well as the latency. Both will be presented in this section. As only one image can be loaded into the circuit at any given time, latency and throughput is directly related. Both depend only on two parameters: the clock frequency and the number of cycles needed to complete the calculation. The clock frequency at various template sizes is known from table 4.1. The number of cycles can be exactly determined by analyzing the state machines in the circuit, together with simulation data for the IP cores.

Figure 4.1 is created from this data. It shows the execution time for the direct implementation of normalized cross-correlation on FPGA as a function of kernel side length for a quadratic kernel.

4.1.4 Power consumption

For a template size of $N = 128$, the Xilinx XPower Analyzer estimates the power consumption for the programmable logic to be 0.69 W, but with a low confidence. This means that this number should only be seen as an indicator of the magnitude of the power consumption. Actual measurements on FPGA can be done to obtain an accurate number.

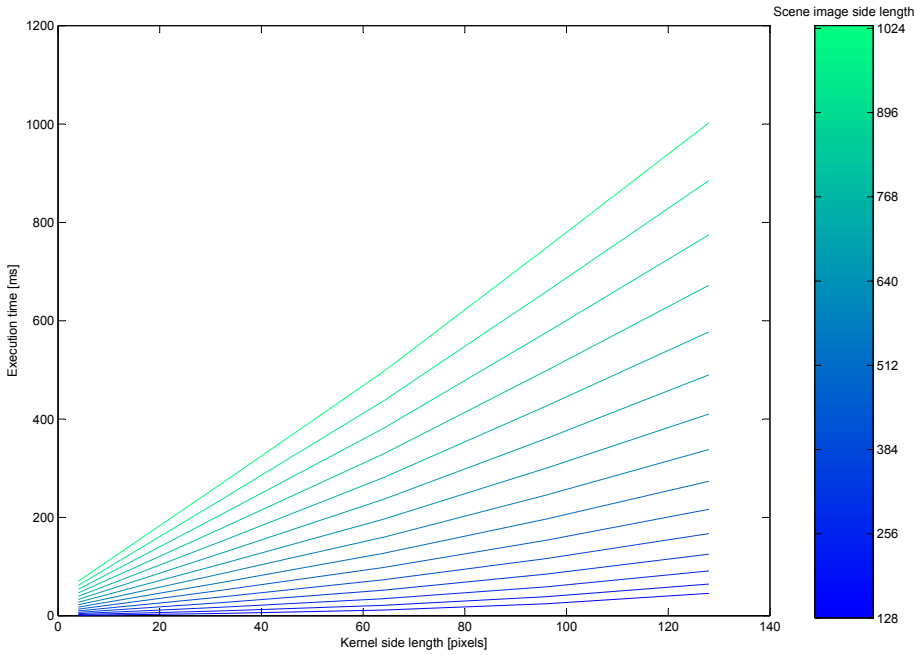


FIGURE 4.1: FPGA performance results. The figure shows execution time as a function of kernel size for various scene image sizes.

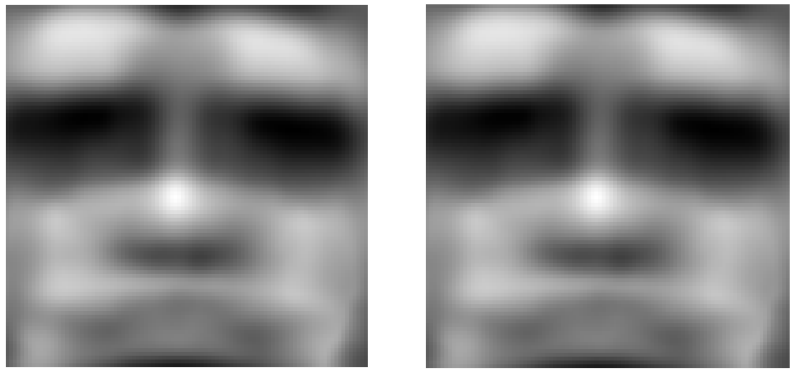
4.1.5 Correctness

Figure 4.3 shows the result of the normalized cross-correlation of the pictures shown in figure 3.8. The blue line shows the results from the MATLAB Image Processing Toolbox `normxcorr2` function, while the red line shows the results of the FPGA implementation. As can be seen from the figure, the error (difference between the lines) is very small. The largest deviation is 0.14 %.

4.2 GPU

4.2.1 Cost and development time

The cost of a CARMA development kit is listed as €529 plus taxes. For product development, the Nvidia Quadro 1000M GPU (or any other GPU) would probably be bought stand-alone at a much lower price, but the exact cost is difficult to verify from public sources.



(A) MATLAB normxcorr2 (B) FPGA implementation

FIGURE 4.2: Results from running normalized cross-correlation in MATLAB and FPGA simulation. A clear bright spot on the man's nose indicates that the normalized cross-correlation is highest in that location.

Development time is very fast when using existing libraries such as OpenCV. It may take a few days to get everything to compile properly for the embedded platform (a guide for doing this is included in appendix A). After this, applications can be programmed in just a few hours to days, taking advantage of the extensive OpenCV library.

4.2.2 Performance

Quadro 1000M

The main result from the performance test of the OpenCV `matchTemplate` function is shown in figure 4.4. The graph shows the GPU execution time (time spent during calculations) as a function of the side length of a square template. The different colors denote different side lengths of the square scene image, in 32 pixels steps from 32 to 1600 pixels.

Figure 4.5 illustrates the unpredictability in execution time for the CARMA GPU implementation. A single data set is used to create the figure. For clarity, only series with spikes (< 10 ms) are included.

Figure 4.6 shows the latency of the direct GPU implementation.

GeForce GTX 660 Ti

Figure 4.7 shows the result of the same test as in figure 4.4, run on a GeForce GTX 660 Ti graphics card. The results are the minimum values of three runs, as

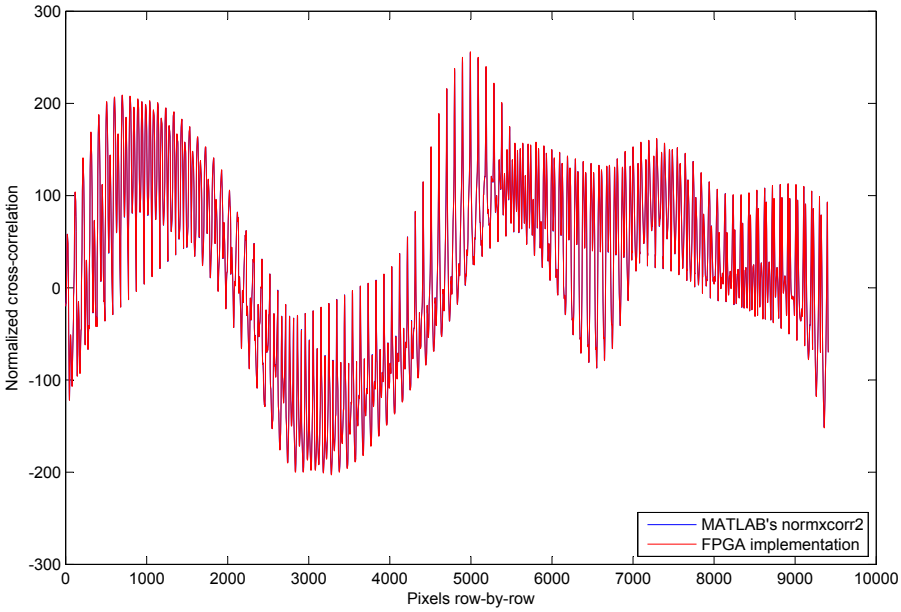


FIGURE 4.3: Comparison between the normalized cross-correlation output from MATLAB's normxcorr2 function and the VHDL implementation designed in this project.

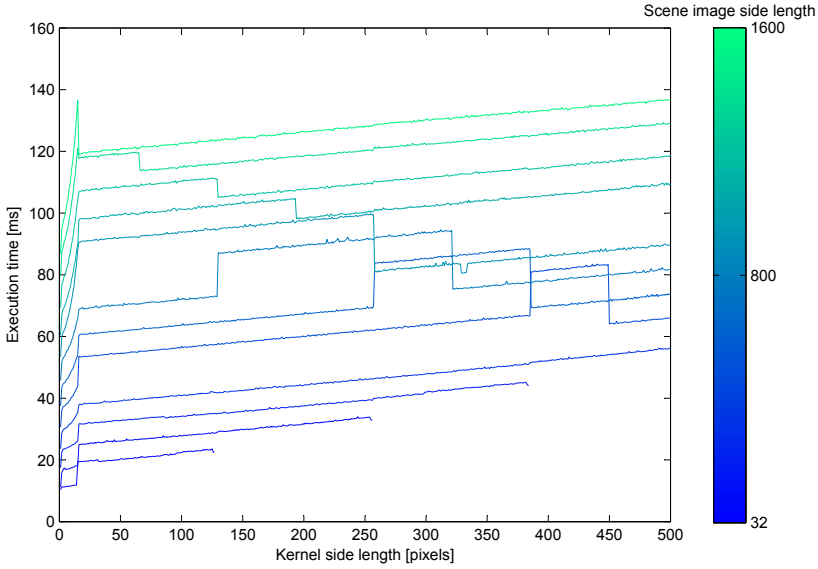


FIGURE 4.4: Latency as a function of kernel side length for a quadratic kernel for varying sizes of a quadratic scene image

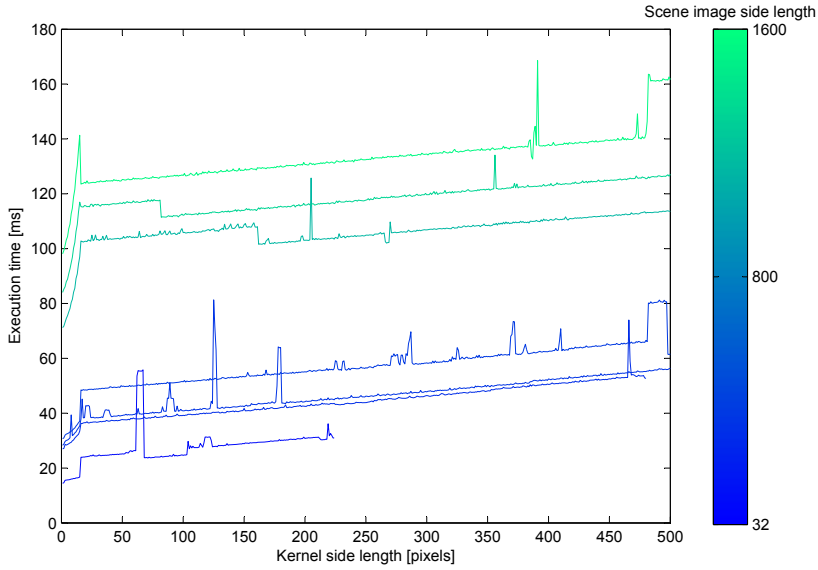


FIGURE 4.5: Variable latency in a single run of GPU execution.

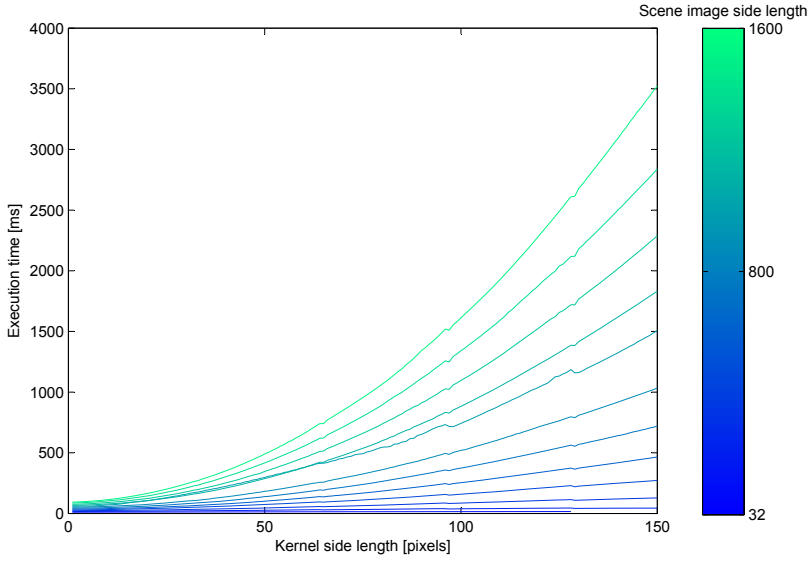


FIGURE 4.6: GPU execution time for the direct implementation of normalized cross-correlation.

in the previous section. The individual runs have large spikes in the processing times, seemingly occurring at random and at different image and template sizes in each run.

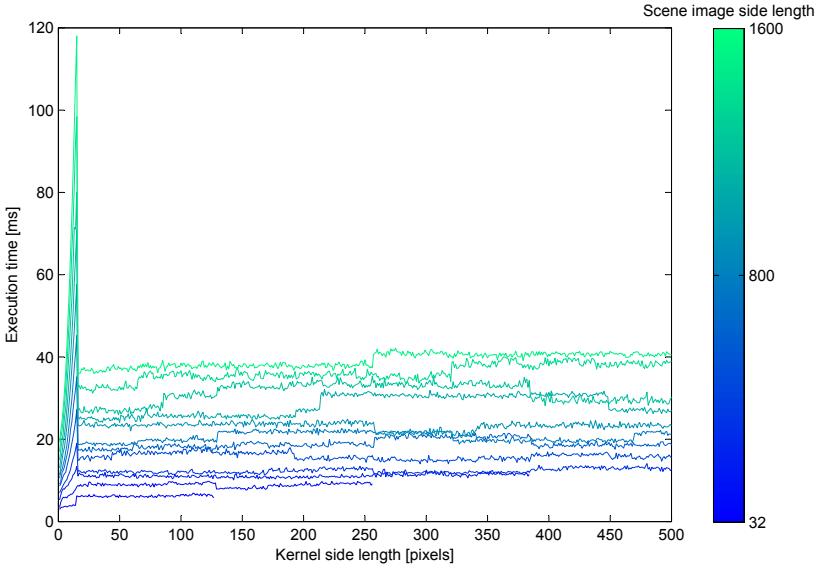


FIGURE 4.7: Latency presented in the same way as in figure 4.4, but from GeForce GTX 660 Ti execution.

4.2.3 Power consumption

The CARMA card consumes 0.51 A (9.69 W) in the idle state and 1.00 A (19 W) when the benchmark test is running.

4.2.4 Correctness

The relevant built-in self-test that comes with OpenCV for testing the `MatchTemplate_CCOEF_NORMED` function passes, as shown in listing 4.1.

LISTING 4.1: Test results from OpenCV built-in self-test

```
[----------] 1 test from GPU_ImgProc/MatchTemplate_CCOEF_NORMED
[ RUN      ] GPU_ImgProc/MatchTemplate_CCOEF_NORMED.Accuracy/0
[       OK ] GPU_ImgProc/MatchTemplate_CCOEF_NORMED.Accuracy/0 (3960 ms)
[----------] 1 test from GPU_ImgProc/MatchTemplate_CCOEF_NORMED (3960 ms total)
```

Chapter 5

Discussion and conclusion

This chapter will summarize the most important results, discuss why they turned out as they did and offer concluding remarks on the choice of technology. Possible areas for further work will also be presented.

5.1 Discussion and further work

The platforms are to be evaluated with respect to development effort, cost, power consumption and performance for image processing algorithms such as normalized cross-correlation. Depending on the specific project or application, some criteria may be more important than others. Some projects may even have absolute requirements, such as throughput in a real-time system or power consumption in an embedded system.

5.1.1 Development effort

For FPGAs, the use of IP can reduce the effort needed, but it is still a time-consuming and sometimes error-prone process to design a complete system. Various high-level tools may somewhat alleviate this. Languages such as HandleC and tools such as Xilinx Vivado HLS and MATLAB HDL Coder and HDL Verifier allow the designer to work at a higher level of abstraction, with automatic generation of HDL code. This comes at the price of higher licensing costs and perhaps sub-optimal solutions, when compared to a manual bottom-up design.

When considering development effort, GPUs have a significant advantage. Open-source libraries such as OpenCV, combined with a relatively high-level language

such as CUDA C++ or OpenCL for custom implementation give short design times. The challenge lies in finding a good partitioning of the problem in to parallel parts, not the programming itself. Software can be tested on any compatible GPU, for example in a desktop computer, even though it is destined for an embedded system. Debugging is seamless and integrated with tools such as NVIDIA Parallel Nsight. There are also many libraries available to make development easier.

5.1.2 Component cost

As pricing information for GPU and FPGA chips is not readily available, an assessment will have to be done before a production project is started, if cost is important.

5.1.3 Power consumption

From the simulations and measurements presented in chapter 4, it is clear that the FPGA solution has significantly lower power consumption than the GPU solution. Because of the uncertainty of the FPGA power simulation, exactly how much is difficult to quantize. The GPU development kit also has a cooling fan, unlike the FPGA kit, which only uses passive cooling.

5.1.4 Performance

The performance results from chapter 4 shows that the GPU implementation outperforms the FPGA implementation. For large image sizes, the GPU implementation use FFT-based correlation, which is very efficient. No comparable algorithm is made for the FPGA in this work because of memory constraints, so these results cannot be directly compared against the FPGA results.

However, the performance of the direct implementation of normalized cross-correlation on FPGA and GPU can be directly compared. The execution time for finding a template in a 512×512 scene image is shown in figure 5.1. This shows that execution time for the GPU implementation is slightly better than for the FPGA implementation.

The DFT-based correlation on the GPU has unpredictable execution time, as shown in figure 4.5. This may be caused by operating system scheduling issues. For real-time systems, this unpredictability may be undesirable. Another issue with the GPU solution is that the first call to the template matching function is very slow, because of Just-in-Time compiling.

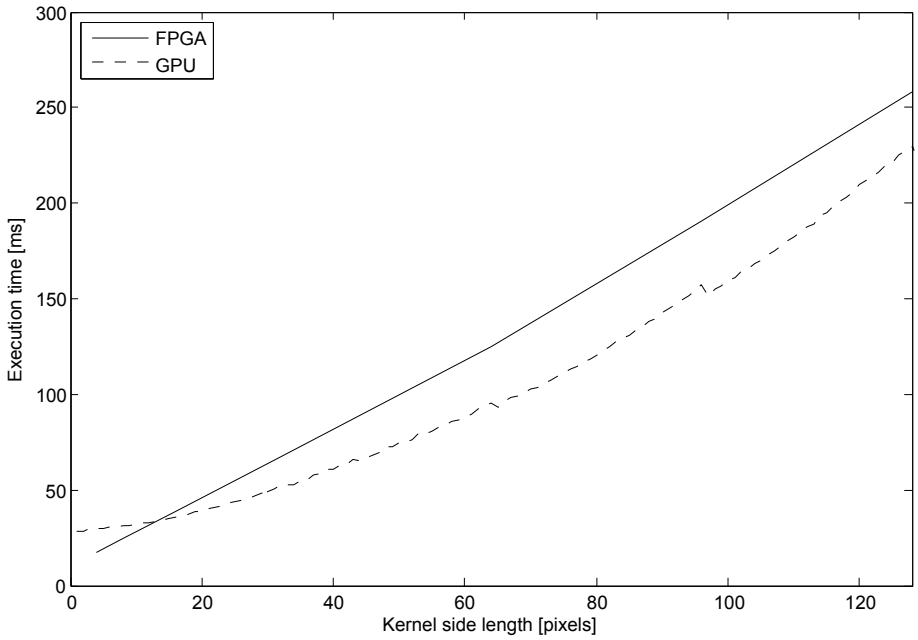


FIGURE 5.1: Execution time for direct implementation of normalized cross-correlation on FPGA and GPU for scene image width $X = 512$.

5.2 Conclusion

In conclusion, it is relatively easy to build a solution with high throughput using a GPU, with the help of available libraries. The reference GPU implementation of normalized cross-correlation from the OpenCV library slightly outperforms the implementation designed in this project. The maximum size of the images processed on the FPGA using this design is dependent on the amount of resources available on the FPGA. For larger images, a more powerful FPGA must be used. The high performance and relative ease of software development of GPUs has to be weighed against the higher power consumption in real-world projects.

5.3 Future work

The most important area of future exploration is DFT-based correlation on FPGA. It is possible that a significant speed-up could be achieved by using DFT-based correlation combined with running sum tables for the normalization on a powerful FPGA with fast off-chip RAM. For large image sizes, DFT-based correlation is necessary if low latencies are critical.

Bibliography

[1] Roberto Brunelli. *Template Matching Techniques in Computer Vision: Theory and Practice*. Wiley, 1 edition, April 2009. ISBN 0470517069.

[2] ADSP-21489 data sheet, 2013. URL `http://www.analog.com/static/imported-files/data_sheets/ADSP-21483_21486_21487_21488_21489.pdf`.

[3] Zynq-7000 all programmable SoC overview. Technical report, August 2012. URL `http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`.

[4] SECO. CARMA devkit technical specs, June 2013. URL `www.seco.com/carmakit`.

[5] Bruce G. Batchelor, editor. *Machine Vision Handbook*. Springer, 2012 edition, January 2012. ISBN 1849961689.

[6] L.M. Russo, E.C. Pedrino, E. Kato, and V.O. Roda. Image convolution processing: A GPU versus FPGA comparison. In *2012 VIII Southern Conference on Programmable Logic (SPL)*, pages 1 –6, March 2012. doi: 10.1109/SPL.2012.6211783.

[7] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 3 edition, August 2007. ISBN 013168728X.

[8] J. P. Lewis. Fast normalized cross-correlation. *Vision Interface*, pages 120–123, 1995.

[9] Durgaprasad Gangodkar, Sachin Gupta, Gurbinder Singh Gill, Padam Kumar, and Ankush Mittal. Efficient variable size template matching using fast normalized cross correlation on multicore processors. In P. Santhi Thilagam, Alwyn Roshan Pais, K. Chandrasekaran, and N. Balakrishnan, editors, *Advanced Computing, Networking and Security*, number 7135 in Lecture Notes in Computer Science, pages 218–227. Springer Berlin Heidelberg, January 2012. ISBN 978-3-642-29279-8, 978-3-642-29280-4. URL `http://link.springer.com/chapter/10.1007/978-3-642-29280-4_26`.

[10] The MathWorks. Normalized 2-d cross-correlation, 2013. URL http://www.mathworks.se/help/images/ref/normxcorr2.html.

[11] *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, September 2007. ISBN 9780521880688.

[12] Xilinx. LogiCORE fast fourier transform v8.0, July 2012. URL http://www.xilinx.com/support/documentation/ip_documentation/ds808_xfft.pdf.

[13] John G. Proakis and Dimitris K. Manolakis. *Digital Signal Processing*. Prentice Hall, 4 edition, April 2006. ISBN 0131873741.

[14] Xilinx. LogiCORE fast fourier transform v8.0 bit accurate c model, September 2010. URL http://www.xilinx.com/support/documentation/ip_documentation/xfft_bitacc_cmodel_ug459.pdf.

[15] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002. ISSN 0360-0300. doi: 10.1145/508352.508353. URL http://doi.acm.org/10.1145/508352.508353.

[16] Xilinx. XST user guide for virtex-6, spartan-6, and 7 series devices, April 2012. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/xst_v6s6.pdf.

[17] Xilinx. LogiCORE IP divider generator v4.0, June 2011. URL http://www.xilinx.com/support/documentation/ip_documentation/div_gen/v4_0/ds819_div_gen.pdf.

[18] J. S. Walther. A unified algorithm for elementary functions. In *Proceedings of the May 18-20, 1971, spring joint computer conference*, AFIPS '71 (Spring), page 379–385, New York, NY, USA, 1971. ACM. doi: 10.1145/1478786.1478840. URL http://doi.acm.org/10.1145/1478786.1478840.

[19] Xilinx. LogiCORE IP CORDIC v4.0, March 2011. URL http://www.xilinx.com/support/documentation/ip_documentation/cordic_ds249.pdf.

[20] Donald G. Bailey. *Design for Embedded Image Processing on FPGAs*. Wiley-IEEE Press, 1 edition, August 2011. ISBN 0470828498.

[21] Xiaotao Wang and Xingbo Wang. FPGA based parallel architectures for normalized cross-correlation. In *2009 1st International Conference on Information Science and Engineering (ICISE)*, pages 225 –229, December 2009. doi: 10.1109/ICISE.2009.603.

[22] NVIDIA. CUDA c programming guide, October 2012. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[23] OpenCV API reference: matchTemplate. URL http://docs.opencv.org/modules/imgproc/doc/object_detection.html?highlight=matchtemplate#cv.MatchTemplate.

[24] OpenCV 2.4.3, February 2012. URL http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.3/OpenCV-2.4.3.tar.bz2/download.

[25] Product comparison quadro mobile series. Technical report. URL http://www.nvidia.com/content/PDF/product-comparison/Product-Comparison-Quadro-mobile-series.pdf.

[26] Martin D. Levine. ECSE-529 image processing assignment #4, August 2007. URL http://www.cim.mcgill.ca/~image529/TA529/Image529_99/assignments/template_matching/template_matching.html.

[27] Xilinx. Xilinx power tools tutorial, March 2010. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug733.pdf.

[28] OpenCV GPU FAQ, June 2011. URL http://opencv.willowgarage.com/wiki/OpenCV%20GPU%20FAQ?action=recall&rev=23#Why_first_function_call_is_slow.3F.

[29] Massimiliano. Setting up a CARMA kit, 2012. URL http://cudamusing.blogspot.no/2012/10/setting-up-carma-kit.html.

[30] Cross-compiling CUDA SDK for CARMA DevKit, 2012.

Appendix A

Setting up the CARMA card

In this chapter we explain how to install the necessary software to compile software for the CARMA development board. The instructions have been adapted from [29, 30] to work on a computer running 64-bit Ubuntu 12.10 with no NVIDIA graphics card installed.

A.1 CUDA SDK samples

We start with a fresh installation of Ubuntu 12.10. Other versions may work, but it must be a 64-bit version. Before starting, download the CUDA Toolkit and CUDA Software Development Kit from `http://www.seco.com/carmakit`. Then, update the package list and install some basic developer tools:

```
$ sudo apt-get update
$ sudo apt-get install freeglut3-dev build-essential libx11-dev
    libXmu-dev libXi-dev libgl1-mesa-glx libglu1-mesa libglu1-mesa-
    dev
```

Install the 32-bit development libraries needed for compiling to the CARMA card:

```
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
$ sudo apt-get install ia32-libs
```

Install the ARM cross compilers. GCC 4.5 is the only version officially supported for targeting the CARMA kit. However, this version is not available for Ubuntu 12.10, and GCC 4.6 was found to cause no problems when compiling the CUDA Samples.

```
$ sudo apt-get install gcc-4.6-arm-linux-gnueabi g++-4.6-arm-linux-
    gnueabi
```

Install the CUDA Toolkit downloaded from `http://www.seco.com/carmakit`.

```
$ sudo sh cuda-linux-ARMv7-rel-4.2.10-13489154.run
```

nvcc should be added to the PATH. This can be done by appending the following line to .bashrc.

```
export PATH=/usr/local/cuda/bin:$PATH
```

Power on the CARMA card and install necessary ARM OpenGL and X11 libraries. Connect the card to your local network and run ifconfig to get the IP address. The card may then be accessed via SSH in the next steps.

```
ubuntu@carma-devkit:~$ sudo apt-get install libglut3-dev libXi-dev
    libXmu-dev
ubuntu@carma-devkit:~$ ifconfig
```

The only purpose of installing the libraries in the previous step was to get the ARM version of the libraries, so that they may be copied to the development PC. When the toolkit was downloaded from `http://www.seco.com/carmakit`, it included a file called copy_from_fs.sh. Copy this file to the directory where the toolkit is installed. Then run it with the IP address of the card found in the previous step as an argument.

```
$ sudo cp copy_from_fs.sh /usr/local/cuda/.
$ cd /usr/local/cuda
$ sudo sh copy_from_fs.sh 192.168.0.100
```

Install the CUDA Software Development Kit downloaded from `http://www.seco.com/carmakit`.

```
$ sudo sh gpu-computing-sdk-linux-4.02.0913.1014-13896433.run
```

Finally, try to compile the samples included in the CUDA SDK.

```
$ cd ~/NVIDIA_GPU_Computing_SDK
$ sudo make ARMv7=1 CUDA_INSTALL_PATH=/usr/local/cuda CXX=/usr/bin/
    arm-linux-gnueabi-g++-4.6
```

If make fails with an error message such as "cannot find -lGL", create a symbolic link to the missing library. This may happen if there is no NVIDIA card or drivers installed on the developer PC. Other libraries in other locations may be needed, the below is just an example.

```
$ cd arm-linux-gnueabi/
$ sudo ln -s ~/NVIDIA_GPU_Computing_SDK/lib/libGL.so.302.26 libGL.
    so
```

A.1.1 OpenCV

Once the samples can be compiled properly, we can compile and install the OpenCV library.

```
$ sudo apt-get install make python python-dev python-numpy git
    libgtk2.0-dev libavcodec-dev libavformat-dev libswscale-dev
    pkgconfig
$ git clone git://github.com/thrust/thrust.git
$ sudo cp -r thrust/thrust <path to cuda toolkit>/include/thrust
```

```
$ cd cmake-2.8.10.2/
$ ./configure
$ make
$ sudo make install
```

```
$ git clone git://code.opencv.org/opencv.git
$ git clone git://code.opencv.org/opencv_extra.git
$ cd opencv
$ mkdir build
$ cd build
$ cmake -DCMAKE_TOOLCHAIN_FILE=../modules/gpu/misc/carma.toolchain.
    cmake -DCUDA_TOOLKIT_ROOT_DIR=/usr/local/cuda -DCUDA_ARCH_BIN:
    STRING="2.1(2.0)" -DCUDA_ARCH_PTX:STRING="" -DCMAKE_SKIP_RPATH:
    BOOL=ON -DWITH_CUBLAS:BOOL=ON ..
$ make
```

Appendix B

A formula suitable for FPGA implementation

The purpose of this appendix is to show how (2.1) can be re-written to be more implementation-friendly.

Let

$$T'(x',y') = T(x',y') - \frac{\sum_{x'',y''} T(x'',y'')}{w \cdot h} \quad \text{(B.1)}$$

$$I'(x+x',y+y') = I(x+x',y+y') - \frac{\sum_{x'',y''} I(x+x'',y+y'')}{w \cdot h} \quad \text{(B.2)}$$

be the template and scene image under the template, both with the respective mean removed. Then, as previously shown, the normalized cross-correlation may be written as

$$R(x,y) = \frac{\sum_{x',y'}(T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}} \quad \text{(B.3)}$$

The numerator may be written out as

$$\sum_{x',y'} \left[T(x',y')I(x+x',y+y') - \frac{T'(x',y') \sum_{x'',y''} I(x+x'',y+y'')}{M \times N} - \frac{I(x+x',y+y') \sum_{x'',y''} T(x'',y'')}{M \times N} + \frac{\sum_{x'',y''} T(x'',y'') \sum_{x'',y''} I(x+x'',y+y'')}{M^2 \times N^2} \right]$$

The last term is constant. Rewriting with this in mind, we get

$$\sum_{x',y'} T(x',y')I(x+x',y+y') - \frac{\sum_{x',y'} T'(x',y') \sum_{x'',y''} I(x+x'',y+y'')}{M \times N} - \frac{\sum_{x',y'} I(x+x',y+y') \sum_{x'',y''} T(x'',y'')}{M \times N} + \frac{\sum_{x'',y''} T(x'',y'') \sum_{x'',y''} I(x+x'',y+y'')}{M \times N}$$

$$= \sum_{x',y'} T(x',y')I(x+x',y+y') - \frac{\sum_{x',y'} T(x',y') \sum_{x',y'} I(x+x',y+y')}{M \times N} \tag{B.4}$$

Using the same approach, the denominator may be rewritten as

$$\sqrt{\sum_{x',y'} T(x',y')^2 - \left[\sum_{x',y'} T(x',y')\right]^2 /(M \times N)} \times \sqrt{\sum_{x',y'} I(x+x',y+y')^2 - \left[\sum_{x',y'} I(x+x',y+y')\right]^2 /(M \times N)} \tag{B.5}$$

Combining this, we get

$$R(x,y) = \frac{MN \sum_{x',y'} (T(x',y')I(x+x',y+y'))}{\sqrt{MN \sum_{x',y'} T(x',y')^2 - \left[\sum_{x',y'} T(x',y')\right]^2}} \cdots \cdots \frac{- \sum_{x',y'} T'(x',y') \sum_{x',y'} I'(x+x',y+y')}{\sqrt{MN \sum_{x',y'} I(x+x',y+y')^2 - \left[\sum_{x',y'} I(x+x',y+y')\right]^2}} \tag{B.6}$$

which only includes the cross-correlation of scene image and template, as well as the sum and squared sum of scene image and template.

Appendix C

Throughput

This appendix contains plots of the throughput of the GPU (figure C.1 and C.2) and FPGA (figure C.3 and C.4) solutions.

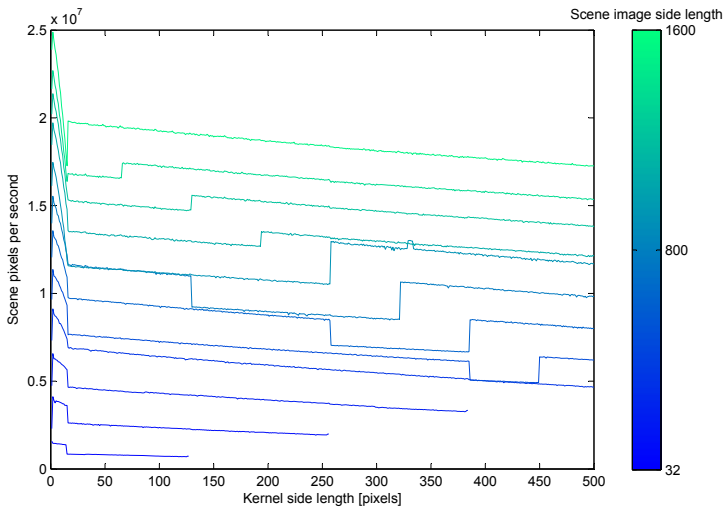


FIGURE C.1: Throughput for GPU as a function of kernel side length for a quadratic kernel for varying sizes of a quadratic scene image.

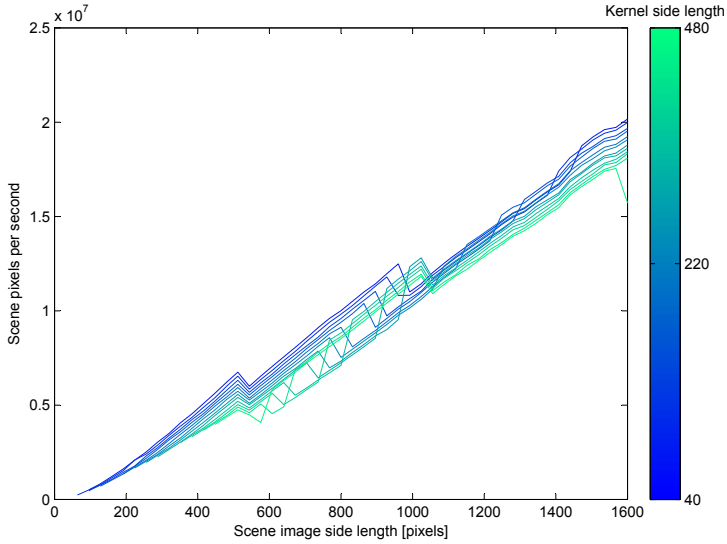


FIGURE C.2: Throughput for GPU as a function of scene image side length.

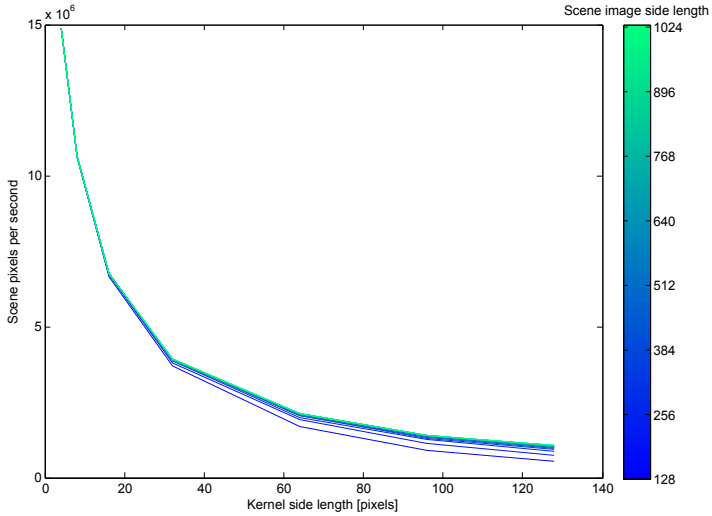


FIGURE C.3: Throughput for FPGA as a function of kernel side length for a quadratic kernel for varying sizes of a quadratic scene image.

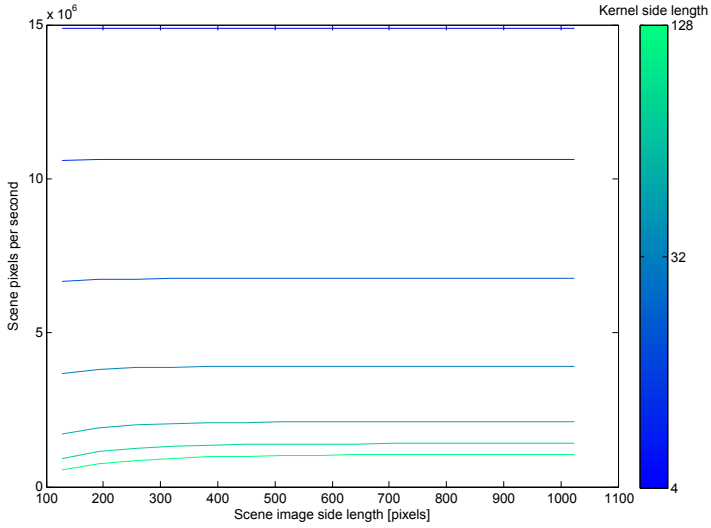


FIGURE C.4: Throughput for FPGA as a function of scene image side length.