FPGA-Accelerated Image Processing Using High Level Synthesis with OpenCL

Johan Isaksson
Master of Science Thesis in Electrical Engineering

FPGA-Accelerated Image Processing Using High Level Synthesis with OpenCL

Johan Isaksson

LiTH-ISY-EX--17/5091--SE

Supervisor:  Erik Bertilsson
Asian, Linköpings universitet

Hans Bohlin
Saab

Examiner:  Kent Palmkvist
Asian, Linköpings universitet

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2017 Johan Isaksson
Abstract

High Level Synthesis (HLS) is a new method for developing applications for use on FPGAs. Instead of the classic approach using a Hardware Descriptive Language (HDL), a high level programming language can be used. HLS has many perks, including high level debugging and simulation of the system being developed. This shortens the development time which in turn lowers the development cost.

In this thesis an evaluation is made regarding the feasibility of using SDAccel as the HLS tool in the OpenCL environment. Two image processing algorithms are implemented using OpenCL C and then synthesized to run on a Kintex Ultra-scale FPGA. The implementation focuses both on low latency and throughput as the target environment is a video distribution network used in vehicles. The network provides the driver with video feeds from cameras mounted on the vehicle.

Finally the test result of the algorithm runs are presented, displaying how well the HLS tool has preformed in terms of system performance and FPGA resource utilization.
## Contents

1 Introduction .......................................................... 1  
   1.1 Motivation ......................................................... 1  
   1.2 Aim ............................................................... 2  
   1.3 Research Questions ............................................... 2  
   1.4 Delimitations ..................................................... 2  

2 Background .......................................................... 3  
   2.1 System Platform .................................................. 3  
      2.1.1 Processing Node Placement .................................. 4  

3 Theory ............................................................. 7  
   3.1 High-Level Synthesis ............................................ 7  
   3.2 OpenCL .......................................................... 8  
      3.2.1 System Overview ............................................ 8  
   3.3 SDAccel .......................................................... 9  
      3.3.1 Compilation Flow .......................................... 9  
      3.3.2 Attributes .................................................. 10  
   3.4 Contrast Enhancement ........................................... 12  
      3.4.1 Histogram Equalization .................................. 12  
      3.4.2 Adaptive Histogram Equalization ......................... 13  
      3.4.3 Contrast Limited Adaptive Histogram Equalization .... 14  
   3.5 Lens Distortion .................................................. 16  
      3.5.1 Pinhole Camera Model .................................... 16  
      3.5.2 Radial Distortion ......................................... 17  
      3.5.3 Correction ................................................ 18  
   3.6 Related work .................................................... 20  
      3.6.1 OpenCL .................................................... 20  
      3.6.2 CLAHE ..................................................... 20  
      3.6.3 Rectification ............................................... 21  

4 Methodology ........................................................ 23  
   4.1 Design Flow ..................................................... 23  
   4.2 Test Platform ................................................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1</td>
<td>FPGA Board</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Camera Parameters</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Evaluation Metrics</td>
<td>27</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Computational Performance</td>
<td>27</td>
</tr>
<tr>
<td>4.4.2</td>
<td>FPGA Resource Usage</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Computational Analysis</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Analysis of CLAHE</td>
<td>29</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Ways of Implementation</td>
<td>29</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Computational Complexity</td>
<td>30</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Data Dependencies</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Analysis of RDC</td>
<td>33</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Ways of Implementation</td>
<td>33</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Computational Complexity</td>
<td>34</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Data dependencies</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>Implementation</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>CLAHE</td>
<td>35</td>
</tr>
<tr>
<td>6.1.1</td>
<td>CDF Kernel</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Interpolation Kernel</td>
<td>38</td>
</tr>
<tr>
<td>6.2</td>
<td>RDC</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>Test Result</td>
<td>41</td>
</tr>
<tr>
<td>7.1</td>
<td>CLAHE</td>
<td>41</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Latency</td>
<td>41</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Area Utilization</td>
<td>43</td>
</tr>
<tr>
<td>7.2</td>
<td>RDC</td>
<td>44</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Latency</td>
<td>44</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Area Utilization</td>
<td>45</td>
</tr>
<tr>
<td>8</td>
<td>Discussion</td>
<td>47</td>
</tr>
<tr>
<td>8.1</td>
<td>Implementation</td>
<td>47</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Design Choices</td>
<td>47</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Latency</td>
<td>48</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Area Utilization</td>
<td>49</td>
</tr>
<tr>
<td>8.2</td>
<td>Exploiting Parallelism with SDAccel</td>
<td>50</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Burst Memory Access</td>
<td>50</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Loop Unrolling</td>
<td>50</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Data Types</td>
<td>50</td>
</tr>
<tr>
<td>8.3</td>
<td>OpenCL</td>
<td>52</td>
</tr>
<tr>
<td>8.3.1</td>
<td>SDAccel</td>
<td>52</td>
</tr>
<tr>
<td>8.4</td>
<td>Node Placement</td>
<td>53</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>55</td>
</tr>
<tr>
<td>9.1</td>
<td>Research Questions</td>
<td>55</td>
</tr>
<tr>
<td>9.2</td>
<td>Future Work</td>
<td>56</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Optimization</td>
<td>56</td>
</tr>
</tbody>
</table>
Contents

9.2.2 New Software Tools .................................................. 56

Bibliography ........................................................................ 57
1

Introduction

This chapter contains the motivation for the thesis, the aim, the questions to be answered and delimitations.

1.1 Motivation

Many applications infer a computational load that is too large for general purpose processors. This problem is often addressed by introducing some kind of hardware accelerator for the specific computational task. Normally these hardware accelerators are developed in a Hardware Descriptive Language (HDL) which can be simulated and later tested on a Field Programmable Gate Array (FPGA). However, a well known problem is that when developing hardware most of the time is not spent on developing new functionality but tends to be spent on troubleshooting basic functionality on a low level. Consequently, the productivity of the development team is reduced. A fairly new approach that addresses this problem is to use High Level Synthesis (HLS) which enables a high level programming language to be used, instead of HDL. HLS translates the high level code to HDL which then can be used for programming the FPGA.

The most prominent feature of HLS is that a software developer with no or little experience of hardware can be assigned to implement an algorithm in C, C++ or OpenCL C and then let a software tool generate the synthesizable code. The development time can then be reduced as the algorithm can be simulated on a high level and less time is needed for troubleshooting low level details.

When using the OpenCL approach the algorithms may not need to be translated with HLS to run on a FPGA but can also be mapped to other platforms supported by the OpenCL framework. This makes it possible to quickly compare how an algorithm performs across different platforms.
1.2 Aim

This thesis aims to address how performance and area utilization is affected when using OpenCL to develop algorithms for use on a FPGA. The algorithms in context are Contrast Limited Adaptive Histogram Equalization (CLAHE) and Radial Distortion Correction (RDC). With statistics acquired from algorithm runs the feasibility of using OpenCL as an acceleration tool can be evaluated.

1.3 Research Questions

The following questions are answered throughout the thesis:

- Are the image processing algorithms suitable for use where low latency is a critical factor?
- Is OpenCL a suitable choice of framework for implementing the algorithms?

1.4 Delimitations

Since the classic OpenCL model builds upon a host processor that passes data to one or more compute devices[1], the focus lie on devices of this type. The only device that receives an implementation is a FPGA, namely the Xilinx KU3 board which is compliant with OpenCL. To program the FPGA Xilinx SDAccel 2016.4 is used.

The thesis covers the image processing algorithms CLAHE and RDC, and all results are derived from the development of the algorithms. The algorithms are implemented in the OpenCL framework where OpenCL C is used for the kernels and C++ for the hosts. All input images for the algorithms are encoded in a 8-bit grayscale format to avoid unnecessary data conversions.

The package based network is realized using Ethernet but is not implemented. The network does only act as a reference point to keep in mind for the implementation of the algorithms.

The resulting data from the algorithm runs that are considered are computational latency and area utilization.
Chapter 2 contains a brief description explaining the background of the thesis. The Video Distribution Network and possible processing node placements are presented.

2.1 System Platform

A basic overview of the Video Distribution System (VDS) is shown in Figure 2.1.

Figure 2.1: Generic VDS network
The system consists of several nodes that are connected using an Ethernet based network. The nodes can be either cameras, displays, data processing nodes or data control nodes. Each node that handles video streams contains a Digital Video Adapter (DiVA) which consist of a FPGA, memory modules and peripherals for communication over ethernet. The FPGA is partly programmed with logic that, depending on if placed inside an camera or display node, compresses/decompresses the video streams before/after they are transferred across the network. It is also programmed with a soft CPU that controls the compression/decompression flow alongside controlling the network communication.

Before the video is transferred from the camera nodes the frames are divided into blocks consisting of 8x8 pixels. The node then packs as many pixel blocks as possible into an Ethernet package. The display nodes receives the packages and updates the display as soon as all pixels for a frame has been retrieved. The user of VDS can watch any display and request a video stream from any of the connected camera nodes.

VDS is most often used for providing user with real time video streams, which in turn infers a requirement of low latency. Latency in this sense is the time from the camera captures a frame until it is shown on one of the displays. Today that latency is around 5ms. Depending on the users mission, the importance of low latency may vary. If the user uses VDS for driving a vehicle, low latency is critical.

2.1.1 Processing Node Placement

Additional processing power in the system is required in order to be able to do more sophisticated image processing. The placement of the processing node and the characteristics of the video processing algorithm may influence the implementation. Figure 2.2 presents the node placements of interest.

Figure 2.2: Reduced VDS network with node placements
The first node placement is inside a camera node, shown as block A in Figure 2.2. This position suits image processing well as algorithms can be applied to the video before it is compressed for network transfer. Additionally, the pixels from the camera can be used in an arbitrary order as it has still not been packed into the 8x8 pixel blocks.

The second alternative is a standalone node, shown as block B in Figure 2.2. This placement is more flexible as it is not a part of any of the existing nodes. However, as the video must be transferred to and from this node additional compression/decompression logic is needed. The implementation of the algorithms is also affected since the pixels arrives in 8x8 blocks.

The third placement is inside a display node, shown as block C in Figure 2.2. This node does already have the decompression logic, but similar to placement B, is affected by the data transfer method. One advantage of placement C is that there is some display nodes in the existing VDS that contains a CPU. This CPU would be able to act as the host for an OpenCL system, which in contrast to the other node placements would not need an additional CPU.
This chapter contains the theory that is needed to successfully follow the rest of the thesis. At the end there are some related works that is of interest when discussing the outcome of the study.

3.1 High-Level Synthesis

The process of synthesis can be described as a translation from a behavioural description of the system to a structural description [8]. Until recent years the behavioural description has been realized using a Hardware Description Language (HDL) which can be synthesized to a FPGA. Today another option is to use High-Level Synthesis (HLS) instead, which means that the behavioural description of the system is implemented in a high-level programming language. This implies that a large step of the development is instead automated by software and developer can in theory go directly from an algorithm to synthesized hardware. It also implies that the structural information that can be expressed using HDL is lost to the programmer which in turn prolongs synthesization time and decreases performance [2].

However, the process of HLS is very similar to the classic synthesization process, only adding a pre-processing step. The high level code is translated to standard HDL which is then used for synthesis. The time consumed for this translation is often considered negligible compared to the actual synthesization [18].

When examining the new synthesis process there are several aspects that are of interest, mainly synthesization time and performance of the synthesized hardware. Performance does in this case include throughput, latency, area utilization and power consumption. As mentioned earlier this thesis only considers latency and area utilization.
3.2 OpenCL

OpenCL is a abbreviation of Open Computing Language which is an open source framework used for general purpose parallel programming[1]. OpenCL can be used to write programs aimed to run at a wide range of processing units ranging from CPUs and GPUs to application specific DSP-processors. This gives the software developer the opportunity to benefit from the power a heterogeneous system may give. OpenCL can also be used in combination with a HLS tool to allow a FPGA as compute device. The chosen HLS tool for this thesis is described in section 3.3.

3.2.1 System Overview

An OpenCL system is visualized in Figure 3.1.

Figure 3.1: The figure shows a system consisting of several compute devices

The system consists of one CPU host which can be connected to one or several compute devices. Compute devices can be ordinary CPUs or more application specific hardware such as GPUs or FPGAs. Each compute device may in turn consist of several compute units (CUs) with underlying processing elements (PEs). A program executed on a PE is called a kernel. The same kernel can be executed on several PEs in parallel. The programming language used to write the kernels
is called OpenCL C which is based on C99 but is modified to suite the device model.

### 3.3 SDAccel

SDAccel is the Xilinx integration of the FPGA platform into the OpenCL environment [18]. It is an IDE based on Eclipse with built-in functions for the OpenCL development flow. In order to use an FPGA as a device in OpenCL the FPGA must be programmed with the static configuration depicted in Figure 3.2.

![Figure 3.2: OpenCL configuration for FPGA.](image)

After configuration, FPGA contains two regions, one region called the OpenCL region and one called the static region. The OpenCL region is where the OpenCL kernels will be programmed to, while the static region is where all interfaces, that are necessary for communicating over PCIe, are stationed. The most interfacing functionality is achieved using Advanced eXtensible Interface (AXI). As the static region is programmed onto the FPGA, it also occupies some of the available resources.

#### 3.3.1 Compilation Flow

There are three different compilation flows available in SDAccel, CPU-Emulation, HW-Emulation and System.

CPU-Emulation runs the code with the CPU as OpenCL platform. This means that the kernel code is compiled for the CPU architecture which in turn means that no information about the hardware is generated. As the code is executed on the CPU there is also no information regarding data transfers between host
and device memory. The only auto generated report that is available after a CPU-Emulation is the timing report which contains information about performance for different parts of the kernel.

HW-Emulation performs HLS and generates HDL code for the kernel. When the program runs the HDL is simulated on the CPU. This simulation provides an estimate of the fully implemented system. Information that is obtained through this flow is again about performance for different parts of the kernel. Additionally, information about FPGA resource utilization and CPU-FPGA data transfers are available.

System compilation performs HLS on the kernel code and generates the same HDL as in HW-Emulation, but then proceeds to generate a netlist and perform the place and route procedure for the FPGA. When the OpenCL program executes the generated bit stream is uploaded to the FPGA.

### 3.3.2 Attributes

As the SDAccel compiler sometimes struggles to find parallelism in the code SDAccel provides several attribute extensions to the OpenCL API. These extra attributes can be inserted into the kernel code to optimize the program. Table 3.1 shows a list of the attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcl_pipeline_workitems</td>
<td>Executes work items in a pipelined fashion.</td>
</tr>
<tr>
<td>xcl_dataflow</td>
<td>Executes the functions inside a loop in a pipelined fashion.</td>
</tr>
<tr>
<td>xcl_pipeline_loop</td>
<td>Executes the instructions inside the following loops body in pipelined fashion.</td>
</tr>
<tr>
<td>xcl_array_partition</td>
<td>Partitions an array over several memory modules.</td>
</tr>
</tbody>
</table>

#### Work Item Pipelining and Data Flow

In the case of using multiple work items in OpenCL, work item pipelining will cause the functions inside the kernel to be executed in a pipelined fashion. This is visualized in Figure 3.3.

The data flow attribute is very similar but will only be applied if the functions are stationed inside the scope of a loop. Data flow does also require a maximum of one work item.

Both work item pipelining and dataflow infer functional level pipelining which in turn causes resources to be better utilized.

#### Loop Pipelining

Loop pipelining, in contrast to work item pipelining, causes the instructions inside the scope of a loop to be executed in a pipeline fashion so that one loop itera-
tion can be completed each clock cycle. This gives an increase in both throughput and utilization of FPGA resources, but at the same time can cause an increase in iteration latency [17].

However, the compiler must always assure functional correctness of the program, which may hinder the pipelining in various situations. The reason behind this is that the compiler is not able to determine whether the directive will break the functional correctness or not. The default action is to stall the pipeline until functional correctness is guaranteed, which can lead to no pipelining at all.

**Array Partitioning**

Depending on size, arrays in OpenCL C are mapped to either registers or BRAM on the FPGA. A problem with BRAM are the limitation of two access ports which limits the amount of data that can be accessed in parallel [18]. The array partition attribute solves this by explicitly defining how many BRAM modules the array should be mapped to and how the data should be arranged. There are three partition types in SDAccel, block partition, cyclic partition and complete partition.

Block partition divides the array into equally sized chunks and maps each chunk to an own BRAM module.

Cyclic partition does also divide the array into equally sized chunks but maps the data differently. The first element from each chunk is mapped to one BRAM module, the second element from each chunk is then mapped to another BRAM module, and so on for all elements in the chunks.

Complete partitioning divides the array element wise. As this approach would utilize the BRAM modules poorly each element is instead mapped to an own register.
3.4 Contrast Enhancement

This section contains the theory and background for the contrast enhancement algorithms.

3.4.1 Histogram Equalization

Histogram equalization (HE) is a method for enhancing the viewing quality of images with low contrast [9]. To accomplish this for a gray scale image the method can be broken down into four steps:

1. Count the amount of pixels associated with each light intensity value i.e. create the histogram.

\[ H(i) = n_i \]  

where \( H \) is the histogram container and \( n_i \) is the number of pixel with intensity \( i \).

2. Calculate the probability of each contrast value appearing in the image i.e. normalize the histogram.

\[ p(i) = \frac{H(i)}{n} \]  

where \( p(i) \) is the probability of intensity \( i \) and \( n \) is the total number of pixels.

3. Calculate the cumulative probability for each contrast value starting with the lowest.

\[ CDF(i) = \sum_{j=0}^{i} p(j) \]  

where CDF is the cumulative distribution function.

4. Round the cumulative probabilities and create a new image where each pixel corresponds to the original image’s cumulative probability in that pixel.

\[ z_{(x,y)} = \text{round}(CDF(u_{(x,y)}) \times L) \]  

where \( z_{(x,y)} \) is the equalized pixel, \( u_{(x,y)} \) is the original pixel and \( L \) is the number of pixel intensities.

With these steps the histogram equalized image is obtained. The benefits of HE is the widened range in contrast, which can be further adjusted by scaling cumulative probability number acquired in step 3. HE is also reversible as there is no lossy compression involved. The method is however not directly applicable to RGB color images as each color channel would be differently equalized. This can be solved by changing color space to, for example YCbCr which consists of one luminance channel and two chrominance channels. HE can then be applied for the luminance channel only, preserving the original color of the image [14].
The major drawback of HE is the amplification of noise that can arise if the image is homogeneous. A solution to this can be to perform contrast limitation which is described in section 3.4.3.

### 3.4.2 Adaptive Histogram Equalization

Adaptive Histogram Equalization (AHE) is an extended version of HE. Instead of creating a histogram for the complete image this method creates several histograms for smaller regions of the image and then uses them to redistribute luminance [14].

![Figure 3.4: An illustration of AHE subregions.](image)

This has the benefit of being able to locally enhance the contrast. The major drawback from HE is amplified in AHE as the contrast range is the same for each subregion of the image. A smaller region leads to a greater chance for pixels having homogeneous light intensity and therefore causes noise being amplified further.
3.4.3 Contrast Limited Adaptive Histogram Equalization

Contrast Limited Adaptive Histogram Equalization (CLAHE) is a special variant of AHE. Unlike AHE, CLAHE has a contrast limiting feature which limits the over-amplifying in homogeneous regions [15]. The contrast limitation is achieved by clipping the histogram bins to a certain threshold value and then redistributing the excess evenly across the histogram. This action does in turn limit the angle of the CDF which is equivalent to limiting the difference in intensities between pixels. The contrast limiting procedure is visualized in Figure 3.5, 3.6 and 3.7.

Figure 3.5: Example histogram.

The dotted line in Figure 3.5 represents the threshold value for the bin clipping.

Figure 3.6: Clipped histogram without redistributed pixels.
3.4 Contrast Enhancement

Figure 3.7: Clipped histogram with redistributed pixels.

As can be seen in Figure 3.7 some bins end up containing more pixels than the threshold after the redistribution. If that is undesired the contrast limiting method can be applied iteratively until no bins exceed the limit.
3.5 Lens Distortion

A common problem that often is found in cameras where cheap lenses are being used is distortions induced by a bad manufacturing process. The most common type of lens distortion is radial distortion, which means that the distortion pattern is radial symmetric around the optical axis. An effect of this is that straight lines appear to be curved.

3.5.1 Pinhole Camera Model

The pinhole camera model is a mathematical description of how a point $P$ in 3-dimensional coordinates is projected onto a 2-dimensional image plane of an ideal pinhole camera. It’s a very simple yet powerful model although things like geometric distortion isn’t accounted for. If the geometric distortion is corrected the model can be used in many applications.

From Figure 3.8 the following equation can be derived:

$$
\begin{bmatrix}
 u \\
 v
\end{bmatrix} =
-\frac{f}{z}
\begin{bmatrix}
 x \\
 y
\end{bmatrix}
$$

(3.5)

where $x$, $y$ and $z$ is the 3D-coordinates, $f$ the focal length and $u$, $v$ the corresponding point in the image plane.
3.5.2 Radial Distortion

Radial distortion can be modelled as

\[
\begin{bmatrix}
  x_d \\
  y_d
\end{bmatrix} = L(r) \begin{bmatrix}
  \tilde{x} \\
  \tilde{y}
\end{bmatrix}
\]

(3.6)

where \((x_d, y_d)\) is the distorted coordinate, \(L(r)\) the distortion factor, \((\tilde{x}, \tilde{y})\) the undistorted image position and \(r\) the distance from the center which often is the principal point. The function \(L(r)\) can be approximated using Taylor expansion

\[
L(r) = 1 + r + \kappa_1 r^2 + \kappa_2 r^3 + \kappa_3 r^4 + \kappa_4 r^5 + \ldots
\]

(3.7)

where \(\kappa_i\) are the distortion coefficients which are part of the camera calibration. It’s clear that the distortion factor only depends on the distance \(r\), that’s where the name radial distortion originate from. [10]

Barrel Distortion

Barrel distortion is the most common distortion pattern. It causes the center of the image to appear more magnified than the perimeter and this magnification decreases nonlinearly with the distance to the center. An example is depicted in Figure 3.9.

![Figure 3.9: Example pattern with Barrel Distortion.](image-url)
Pincushion Distortion

In contrary to barrel distortion pincushion distortion will cause objects to appear more magnified closer to the perimeter, as depicted in Figure 3.10.

Figure 3.10: Example pattern with Pincushion Distortion.

3.5.3 Correction

The problem of radial distortion can removed through image rectification. Rectification means that one or more images are projected to a common plane through a transformation function [10]. For radial distortion this done with (3.6). For calculating the distortion parameters the distorted image must have a pattern that can be transformed to match a reference pattern. A reference pattern for correcting Figure 3.9 and 3.10 is shown in Figure 3.11.

Figure 3.11: Example pattern without any radial distortions.
Camera Calibration

To correct the distortion the parameters for the camera model can be retrieved using the Matlab Camera Calibrator. The camera calibrator uses a chess pattern as reference and takes a number of images as input and outputs the desired parameters. An example of input image can be seen in Figure 3.12.

*Figure 3.12: Example image for Matlab Camera Calibrator.*
3.6 Related work

This section contains works related to the content of the thesis.

3.6.1 OpenCL

Regarding OpenCL an aspect of interest is the difference in resulting hardware between the OpenCL compiled kernel and a handcrafted HDL version. Also the difference in design flow and development time is interesting when evaluating the development flow.

The authors of [2] demonstrates that a handcrafted HDL design of the Sobel filter outperforms designs generated from kernels using OpenCL when it comes to execution time and chip area consumed. They also concluded that the time consumed to develop the handcrafted version was far longer than for the design generated from OpenCL. The authors of [11] evaluates the same algorithm and states that the area consumed is 59% to 70% less for the handcrafted version but the performance is equal. They noted a six time increase in productivity when using OpenCL. In a similar study, the authors of [3] compared the development time of Fractal compression which is a technique for image and video encoding. They claim that the handcrafted HDL version took a month meanwhile the version generated from OpenCL were up and running within hours. The handcrafted version lacked essential parts such as PCIe and DDR interfaces which comes for free when using vendor specific SDKs to go from OpenCL to RTL.

These related works emphasizes the use of OpenCL if the system under development does not have strict requirements in terms of area. The FPGA board used in this thesis will have a sufficient amount of LUT’s, Block RAMs and logic blocks to handle large systems, but area is still an important aspect which may limit the use of OpenCL.

3.6.2 CLAHE

The authors of [7] concludes that CLAHE is well suited for implementation on an FPGA due to the available block RAMs for histogram storage. This minimizes the costly external memory accesses. However, the article also discusses problems with the algorithm that are seen on other platforms than just FPGA. An example of this is the trade off between histogram size, image quality and memory constraints. The authors of [12] achieved real-time processing of CLAHE on images with a resolution of 1920x1080 by implementing a modified sliding window technique. Their implementation used the previous windows CDF to remap the pixel intensity. This induces an error which they considered small enough to not significantly affect the visual quality of the resulting image.

From these two works it seems that a version of CLAHE that both follows the correct flow and produces good viewing quality may be too computationally demanding even for a FPGA when the image size increases. Some trade offs must be made to achieve the desired result.
3.6.3 Rectification

The authors of [13] present a real-time implementation for distortion removal. By the use of BRAMs for intermediate image storage they are able to apply rectification on two separate images from a stereoscopic camera simultaneously. Their design strongly depends on the rectification parameters obtained in advance, as they in their approach determines how many pixel lines from the image that must be read before rectification can start. This is a limitation that may need to be considered to achieve real-time speed. A dynamic system that can handle any sort of rectification parameters might be too complex or slow for being considered for implementation. The authors of [5] handle the limitation presented in the previous work by decoupling the remapping function from the surrounding hardware, allowing easily exchange of mapping models. This approach has its limitations as they had to use sub sampling to be able to fit the input LUT into memory. To get a pixel value they used bilinear interpolation which will, as they concluded, achieve poor image quality in certain cases.
This chapter contains the methods used to achieve the results of the thesis.

4.1 Design Flow

The first thing that was developed was naive implementations of the algorithms, targeted for use on a CPU. It did not use any particular design flow as it mostly followed pseudo code. It was also solely carried out as starting point for the implementation of FPGA, and as a measure to address the performance of the compiler for FPGA.

The design flow when using OpenCL in general refers to the system seen in Figure 3.1. The amount of devices, compute units and process elements was considered and used as a reference when writing the kernels. On a CPU or GPU each process element will run an instance of the kernel allowing true parallel execution. Depending on the global/work group size the program will automatically utilize the sufficient amount of PEs. However, this is not the case when using a FPGA as device as there are no predefined process elements. The FPGA is rather seen as a blank computational canvas [18] where the functionality for the complete program will be implemented as one single OpenCL PE. This means that a global/work group size larger than one will cause the work items to be executed in a serial manner. Fortunately there is the work item pipeline directive described in section 3.3 which can be used to better utilize the hardware and run different parts of the kernel in parallel. However, this will never be as fast as the true parallel execution of work items seen in GPUs, and as a consequence Xilinx does recommend using a global size of one for maximum performance [17].

When using the Xilinx SDAccel the most time consuming part, i.e. the bit-level verification of the system known in regular HDL development. The system was realized using one or multiple kernels described using the OpenCL C syntax.
The IDE makes it possible to assess the functional correctness of the design by executing it on an emulation device on the CPU host as mentioned in section 3.3. A debug tool, similar to those used during software development, helps to locate the origin of functional errors. When the design was functionally correct, the system compiled with the FPGA as target. The process translated the OpenCL C code to HDL and then synthesized the HDL to generate the bit stream. The process was performed solely by a software tool which guarantees functional correctness of the system.

4.2 Test Platform

The node can be structured in various way to better fit the different placements, but as the scope for this thesis is to evaluate OpenCL all IPN will have a common architecture. Our node has the architecture visualized in Figure 4.1.

![Architecture of the Image Processing Node.](image)

The architecture consists of a computer with an x86 cpu, a Nvidia graphics card and a Xilinx FPGA. Both the FPGA and the GPU are connected to CPU via the PCI express interface. An expected set-up in this case might be to have an external FPGA board, but having it installed internally makes the programming sequence much faster and the the number of connection problems are reduced.

The architecture seen in Figure 4.1 does also allow fast switching between target architectures in OpenCL. This in turn will reduce the start-up time for the algorithm testing.
4.2 Test Platform

4.2.1 FPGA Board

The algorithms in this thesis run on the ADM-PCIe-KU3 FPGA board from Alpha Data. The board is based on the Kintex Ultrascale XCKU060-2 FPGA and contain a PCI express Gen3 8x interface, two 8GB DDR3 ECC-SODIMM modules, 1GBit of BPI x 16 configuration flash, two QSFP cages for high speed ethernet and two SATA interfaces. Figure 4.2 shows a block diagram with the most important components.

![Figure 4.2: Components of the KU3 board.](image)

The FPGA has a standard clock frequency set to 250 MHz and the available resources, consisting of 18Kb Block Random Access Memories (BRAM), Digital Signal Processing blocks (DSP), Flip-Flops (FF) and Look-Up Tables (LUT), are displayed in table 4.1. Table 4.1 presents the amount of each resource.

<table>
<thead>
<tr>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2160</td>
<td>2760</td>
<td>663360</td>
<td>331680</td>
</tr>
</tbody>
</table>
4.3 Camera Parameters

The parameters retrieved by the Matlab Camera Calibrator are presented in matrix form in (4.1).

\[
C = \begin{bmatrix}
  f_x & s & x_0 \\
  0 & f_y & y_0 \\
  0 & 0 & 1 \\
\end{bmatrix} = \begin{bmatrix}
  0.645752996 & 0 & 0.490731266 \\
  0 & 1.146339203 & 0.476013624 \\
  0 & 0 & 1 \\
\end{bmatrix}
\] (4.1)

where \( f_x, f_y, x_0 \) and \( y_0 \) are normalized values. The constants from (3.7) are presented in (4.2).

\[
\kappa = \begin{bmatrix}
  \kappa_1 \\
  \kappa_2 \\
  \kappa_3 \\
\end{bmatrix} = \begin{bmatrix}
  -0.3623 \\
  0.2315 \\
  -0.1081 \\
\end{bmatrix}
\] (4.2)

A total of 40 images were used for the parameter calculation, each with a slightly altered point of view to give the best estimate.
4.4 Evaluation Metrics

In order to be able to evaluate the resulting implementations some metrics are needed. Those of interest are presented throughout this section.

4.4.1 Computational Performance

The computational performance of interest is mainly latency for different parts of the algorithm.

Frame Latency

Frame latency is defined as the time for processing an entire image, i.e. the total runtime for an OpenCL compute unit. Furthermore, it is the total time from start to end of an algorithm run, excluding overhead from the OS and other events that interrupts the process. To measure frame latency the embedded profiler available in SDAccel was used when measuring for the FPGA, and built in performance counters when measuring for the CPU.

Pixel Latency

Pixel latency for the algorithm can be seen as the time from pixel input to pixel output, i.e. the arrival of one pixel until the processing of the pixel is complete. The pixel latency of an algorithm can also be measured using performance counters but that approach will not be used in this study. Performance counters would require integration into the algorithm implementations and most likely make the code untidy. However, using embedded profiler on FPGA clock cycle counts for some internal functions may be available which could yield an estimate.

4.4.2 FPGA Resource Usage

Resource usage is the amount of resources the system will occupy on the FPGA. The resources involve here are Look-Up Tables (LUT), Block RAMs (BRAM), registers consisting of Flip-Flops (FF) and Digital Signal Processing blocks (DSP). A table containing the amount of resources available can be seen in Table 4.1. The numbers are retrieved from the HLS reports that SDAccel generates when compiling the code with FPGA as target platform.
This chapter presents an analysis of the two algorithms, CLAHE and RDC.

5.1 Analysis of CLAHE

From the descriptions of CLAHE mentioned in Chapter 3 there are two main ways to implement the algorithm. This section compares the two ways at a coarse level to justify the choice of implementation.

5.1.1 Ways of Implementation

As can be seen in the brief description of CLAHE the algorithm is straightforward and can seem to demand less computational power in comparison to other image processing algorithms [16]. However, the computational demand arise when the method is to be applied on larger images or video streams in real-time.

Block based CLAHE

The most basic form of CLAHE divides the image into equally sized regions, henceforth called blocks, and performs HE on each of the blocks. This induces a clearly visual but undesired tiling effect in the resulting image. To remove these effects interpolation between the neighbouring block histograms can be used. The most common interpolation strategy is to use bilinear interpolation which uses the euclidean distance between the pixel and the center points of the neighbouring blocks. To achieve a satisfying result many neighbouring blocks may have to be used in the interpolation, which can results in a heavy computational load [6].
Sliding Window based CLAHE

One way to avoid the requirement of interpolation needed for the block based CLAHE is to use a sliding window approach instead.

In each step a histogram is created from a window around the pixel to be equalized. Only the center pixel equalized and when sliding one pixel, the histogram can be incrementally updated, i.e. only the pixels in the sliding direction has to be added to the histogram and pixels behind removed. This approach makes it possible to skip the interpolation since the windows are overlapping.

Using a sliding window, the number of calculations for each window are fewer than for each tile in the block based version of CLAHE. However, each pixel requires their own histogram which drastically increases the number of histograms [15].

5.1.2 Computational Complexity

Both versions of the CLAHE algorithm can be described using the steps from the description of HE, see chapter 3, with the addition of contrast limiting. In this section the sub operations of the algorithm will be noted according to Table 5.1.

<table>
<thead>
<tr>
<th>$T_{op}$</th>
<th>$N_{op}$</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$N_1$</td>
<td>Inserting one pixel into a histogram.</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$N_2$</td>
<td>Clipping one histogram bin.</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$N_3$</td>
<td>Redistributing the overflow into one histogram bin.</td>
</tr>
<tr>
<td>$T_4$</td>
<td>$N_4$</td>
<td>Calculating the cumulative possibility for one histogram bin.</td>
</tr>
<tr>
<td>$T_5$</td>
<td>$N_5$</td>
<td>Interpolating 4 CDF bins.</td>
</tr>
<tr>
<td>$T_6$</td>
<td>$N_6$</td>
<td>Calculating the final light intensity value for one pixel.</td>
</tr>
</tbody>
</table>

Where $T_{op}$ is the time consumed for the operation and $N_{op}$ is the total number of times the operation is used. All operations in Table 5.1 have an computational complexity of $O(1)$. The time variable can be used to show how many operations there are and the computational complexity of the implementation can then be analysed by looking at the number of operations.

Block based CLAHE

The formula in (5.1) describes the time consumed using the block based CLAHE.

$$T_B = \sum_{n_b=0}^{N_b} \left( \sum_{n_pb=0}^{N_pb} T_1 + \sum_{n_cd=0}^{N_cd} \left( T_2 + T_3 + T_4 \right) + \sum_{n_pb=0}^{N_pb} \left( T_5 + T_6 \right) \right)$$  \hspace{1cm} (5.1)

where $T_B$ is the total time consumed by the block based CLAHE, $N_b$ is the number of blocks in one image, $N_{pb}$ is the number pixels in one block and $N_{cd}$ is the number of light intensities.
As all of the steps has constant complexity the complexity of the block based CLAHE is $O(N)$, where $N$ is the number of pixels. Moreover, in the matter of operations the resulting numbers can be seen in (5.2) and (5.3).

$$N_1 = N_5 = N_6 = N_{pb}N_b = N$$ \hspace{1cm} (5.2)

$$N_2 = N_3 = N_4 = N_{cd}N_b$$ \hspace{1cm} (5.3)

This is expected as each pixel must be added to a histogram and equalized with interpolation. It is also expected that there is $N_{cd}$ histogram clipping operations in each block as each block has an independent histogram.

Numbers of the operations with certain image sizes can be seen in Table 5.2.

**Table 5.2: Number of operations with $N_{pb} = 8 \times 8$ and $N_{cd} = 256$**

<table>
<thead>
<tr>
<th>Operation</th>
<th>$N = 1280 \times 720$</th>
<th>$N = 1600 \times 900$</th>
<th>$N = 1920 \times 1080$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_1$</td>
<td>921600</td>
<td>1440000</td>
<td>2073600</td>
</tr>
<tr>
<td>$N_2$</td>
<td>3686400</td>
<td>5760000</td>
<td>8294400</td>
</tr>
</tbody>
</table>

**Sliding Window based CLAHE**

In (5.4) the total amount of operations using the sliding window based CLAHE is described.

$$T_{S_{\text{total}}} = \sum_{n=0}^{N} \left( \sum_{n_{pb}=0}^{N_{pb}} T_1 + \sum_{n_{cd}=0}^{N_{cd}} \left( T_2 + T_3 + T_4 \right) + T_6 \right)$$ \hspace{1cm} (5.4)

Where $T_{B_{\text{total}}}$ is the total time consumed by the sliding window based CLAHE. By updating the histogram incrementally when sliding in the x dimension, the formula can be reduced:

$$T_{S_{\text{total}}} = \sum_{n=0}^{N} \left( 2N_{pb_{H}} T_1 + \sum_{n_{cd}=0}^{N_{cd}} \left( T_2 + T_3 + T_4 \right) + T_6 \right)$$ \hspace{1cm} (5.5)

where $N_{pb_{H}}$ is number of pixels on the height in a block. The formula results in the following number of operations:

$$N_1 = 2N_{pb_{H}}N$$ \hspace{1cm} (5.6)

$$N_2 = N_3 = N_4 = N_{cd}N$$ \hspace{1cm} (5.7)

$$N_6 = N$$ \hspace{1cm} (5.8)

As can be seen, both $N_1$ and $N_2$ are increased. Compared to the block based CLAHE, sliding window CLAHE requires an histogram for each pixel. After utilizing incremental updates of the histogram the filling procedure is reduced from $N_{pb}N$ to $2N_{pb_{H}}N$ as the overlap in the x-dimension is eliminated.
However, even after updating the histogram incrementally the sliding window still overlaps pixels in the y-dimension. Numbers of operations with certain image sizes can be seen in Table 5.3.

**Table 5.3: Number of operations with \( N_{pb} = 8 \times 8 \) and \( N_{cd} = 256 \)**

<table>
<thead>
<tr>
<th>Operation</th>
<th>( N = 1280 \times 720 )</th>
<th>( N = 1600 \times 900 )</th>
<th>( N = 1920 \times 1080 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_1 )</td>
<td>14745600</td>
<td>23040000</td>
<td>33177600</td>
</tr>
<tr>
<td>( N_2 )</td>
<td>235929600</td>
<td>368640000</td>
<td>530841600</td>
</tr>
</tbody>
</table>

From the examples in Table 5.3 and Table 5.2 it can be stated that the block based CLAHE implies a significantly smaller computational load than the sliding window based CLAHE. For the specific example with a block size of \( 8 \times 8 \), the block based CLAHE has 6.25\% the amount of histogram fills and 1.56\% the amount of histogram clipping.

### 5.1.3 Data Dependencies

For the block based CLAHE the dependencies arise during the interpolation part as each tile requires the neighbouring CDFs. Assuming that the algorithm should be implemented as streamlined as possible, in this case one row of blocks at a time, the interpolation and equalization of pixels in the first row must wait for the second row to finish their CDF calculation.

A solution which eliminates this dependency is to use the previous frames CDF. This does however induce an error in the equalization process if the pixel intensities in the block differs between the frames. A scenario of the problem is depicted in 5.1 and 5.2.

![Figure 5.1: Block histogram for previous frame.](image1)

![Figure 5.2: Block histogram for current frame.](image2)

In the scenario above the pixels in the current block would all be darker after the equalization as the previous CDF would be low for the major part.

The effect of incorrectly remapped pixels is attenuated by the interpolation process, as more pixels are involved, and by histogram clipping. Furthermore, the effect can also be attenuated by combining the equalized image with the original image, but that attenuates the histogram equalization as well.
For the sliding window based CLAHE there are no dependencies between neighbouring regions.

## 5.2 Analysis of RDC

This section analyses the correction method of radial distortion.

### 5.2.1 Ways of Implementation

Since the result of (3.6) may result in a fractional number interpolation can be used to retrieve the correct pixel value. For this thesis there are two ways of retrieving the pixel.

**Bilinear Interpolation**

The first and most accurate way is by using bilinear interpolation. It works by linearly interpolating the values between four pixels according to (5.9):

\[
p_{BI} = (1 - \alpha)(1 - \beta)p(x, y) + \alpha(1 - \beta)p(x + 1, y) + (1 - \alpha)\beta p(x, y + 1) + \alpha \beta p(x + 1, y + 1)
\]

where \( p(x, y) \) is the pixel value at position \((x, y)\). \( \alpha \) and \( \beta \) are the distances to top left pixel in the x- and y-dimension respectively, visualized in Figure 5.3.

\[5.9\]

*Figure 5.3: Variables for bilinear interpolation.*
where the four squares each represents a pixel. This approach requires up to four pixels to be fetched from the memory.

**Nearest Neighbor**

Instead of using bilinear interpolation a method known as nearest neighbour may be used where the nearest pixel to the result from 3.6 is used as the result. This reduces the demands on memory structure as only one pixel must be fetched.

### 5.2.2 Computational Complexity

The computational complexity for the radial distortion correction is straightforward.

**Table 5.4:** Description of operations for analysis of RDC.

<table>
<thead>
<tr>
<th>$T_{op}$</th>
<th>$N_{op}$</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$N_1$</td>
<td>Calculating the distorted pixel coordinate.</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$N_2$</td>
<td>Interpolating to retrieve desired pixel value.</td>
</tr>
</tbody>
</table>

Where $T_{op}$ is the time consumed for the operation and $N_{op}$ is the total number of times the operation is used. The total time taken for processing an image can then be described with the following equation:

$$ T_{total} = \sum_{n=0}^{N} \left( T_1 + T_2 \right) $$

(5.10)

which leads to a computational complexity equal to $O(N)$. This is true for both interpolation techniques. However, if the memory structure only allows on pixel to retrieved each cycle, the bilinear interpolation might take up to four times as long as the nearest neighbour version.

### 5.2.3 Data dependencies

As there are no direct dependencies between pixels in the radial distortion correction algorithm, all pixels can in theory be processed in parallel. This is not possible in practice as there is other limiting factors such as number of processing units and memory bandwidth. There is however a situation that hinders the process. The distorted pixel coordinate may be far away. If the algorithm is implemented in a system similar to the network described in Chapter 2, the pixels must be sent over Ethernet and may not be available yet. With the camera parameters from (4.2), and an image size of $N = 1920 \times 1080$, the distorted coordinate can be up to 120 pixel lines away. This means, in the worst case scenario, that the process must wait for 120 lines of pixels before starting, inducing a significant delay.
This chapter contains the implementations and system descriptions for the two algorithms.

### 6.1 CLAHE

The implementation of the block based CLAHE was separated into two kernels. The first kernel generates the histogram, limits the contrast and calculates the CDF for each block. This kernel is henceforth called the CDF kernel. The second kernel uses the calculated CDFs from the first kernel to calculate the final value for each pixel in the image. This other kernel is henceforth called the interpolation kernel. An overview of the implementation is presented in Figure 6.1.

![Figure 6.1: System overview of block based CLAHE.](image)

The block size was set to $64 \times 8 = 512$ pixels. The block width was fixed but the height could have been altered.
6.1.1 CDF Kernel

A flowchart of the CDF generating kernel is visualized in Figure 6.2.

![Flowchart of the CDF calculation in the block based CLAHE.](image)

**Figure 6.2:** Flowchart of the CDF calculation in the block based CLAHE.

The aim of the design is to calculate the CDF for each block in a row in parallel. The input cache is realized using an array of the `uint16` data type from OpenCL. This to make the burst logic simple as an `uint16` equals 64 bytes, matching the pixel width of the blocks. Before the histogram calculation starts the `uint16`s are separated into byte arrays so that each parallel histogram processing unit can retrieve one byte from its corresponding byte array simultaneously. When all 64 pixels in the byte array are inserted into the corresponding block histogram the next pixel line of the blocks are inserted into the byte arrays. This is repeated 8 times, equal to the pixel height of the blocks. Afterwards, all blocks in a row has a complete histogram.
The next step is the contrast limitation. The bins in a block histogram are clipped in a serial manner, i.e. one clipped bin per clock cycle. However, each block in the row has an individual clipping unit which means that pixels are clipped in parallel. The last step is the redistribution and CDF calculation, combined into one action. The total number of pixel clipped from the original histogram, called redistribution amount, is divided by the number of bins in the histogram. The CDF calculation is realized using a loop that iterates over all bins. At each iteration the divided redistribution amount is added to an accumulation register. If the register contains a value larger than 1 the integer part will be moved and added to the current CDF bin. Consequently, the fractional part remains in the accumulation register to later be added to another CDF bin. This method redistributes any amount of pixels evenly over the CDF to a low cost of FPGA resources. As for previous steps, this operation is done in parallel for each block. When all steps are completed, the CDFs are shifted into a shift register for meantime storage. The shift register is realized using the OpenCL pipe data type which contains blocking read and write functions. The blocking write function used in this kernel avoids overflow in the shift register by stalling the kernel until there is room for another write. The pixels from the input cache are written into another pipe to prevent an additional reads from the global memory.
6.1.2 Interpolation Kernel

A flowchart of the interpolation kernel is visualized in Figure 6.3.

![Flowchart of the interpolation in the block based CLAHE.](image)

The aim here is again to process each block in a row in parallel. This kernel starts by reading one row of block CDFs from the shift register with a blocking read function. This prevents the kernel from reading if the shift register is empty and consequently synchronizes the kernels. Each time a row of CDFs has been received the kernel can start the interpolation and pixel remapping. This is however not possible for the first block row in the image due to that the interpolation needs the neighbouring blocks CDF, as mentioned in chapter 5. When the second row of CDFs has been received the pixel remapping of the first row can start.
To reveal parallelism for the compiler the interpolation is divided into four parts, one for each quadrant in the blocks. Furthermore, the fetching of CDF values are separated. This is depicted in Figure 6.4.

![Figure 6.4: Interpolation procedure.](image)

All blocks in a row are simultaneously reading a bin from the top left CDF, then a bin from the top right, and so on. When all four bins have been retrieved they are interpolated based on the pixel position inside the block and then stored in a temporary array. When all pixels from each block are processed they are written to the output cache which in turn is burst written to global memory. The input image and the output image are stored onto separate DDR banks of the board. This makes it possible to read and write to the global memory at the same time.

### 6.2 RDC

The Radial Distortion Correction algorithm is implemented in a simpler manner as the algorithm in general is less complex. The flow of the implementation is visualized in Figure 6.5.

To begin with, a chunk of the image is transferred from the global memory to a local cache. Then, the distorted coordinate of a pixel \(i\) calculated, retrieved from the local cache and stored in the output cache. The flowchart demonstrates a sequential execution, i.e. one pixel per cycle, but the inner loop can in theory be executed in parallel. This is however not possible in this case as it would imply a memory structure too complex for the compiler to realize. When the output cache is full the pixels are transferred back to the global memory. This implementation does also use two DDR banks so that global memory can read from and written to in parallel.
Figure 6.5: Flowchart of the RDC kernel.
This chapter contains the test results for the two algorithms. Section 7.1 contains the test results for the CLAHE implementation and section 7.2 contains the test results for the RDC implementation.

7.1 CLAHE

This section presents performance numbers and resource usage for both kernels described in section 6.1.

7.1.1 Latency

To begin with the overall frame latency is presented. The two kernels are then further broken down into smaller parts in order to verify what function is the most time consuming.

Table 7.1: Frame latency for different implementations of CLAHE using CPU as OpenCL platform.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Latency [ms]</th>
<th>( N = 384 \times 280 )</th>
<th>( N = 640 \times 480 )</th>
<th>( N = 1280 \times 720 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU model</td>
<td>6.11</td>
<td>16.87</td>
<td>48.82</td>
<td></td>
</tr>
<tr>
<td>CPU port</td>
<td>3.93</td>
<td>12.21</td>
<td>33.68</td>
<td></td>
</tr>
<tr>
<td>FPGA</td>
<td>7.67</td>
<td>25.39</td>
<td>83.29</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1 demonstrates the difference in frame latency between the different implementations of the algorithm when using the CPU as OpenCL platform.
CPU port is the CPU model with minor adjustments to allow synthesization for the FPGA. FPGA refers to the implementation described in chapter 6.

Table 7.2: Frame latency for different implementations of CLAHE using FPGA as OpenCL platform.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Latency [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N = 384 \times 280$</td>
</tr>
<tr>
<td>CPU port</td>
<td>-</td>
</tr>
<tr>
<td>FPGA</td>
<td>3.61</td>
</tr>
</tbody>
</table>

Similar to Table 7.1, Table 7.2 demonstrates the difference in frame latency between the different implementations of the algorithm when using the FPGA as OpenCL platform. SDAccel was not able synthesize the CPU port with the available resources.

Table 7.3: Clock cycle cost of CDF kernel.

<table>
<thead>
<tr>
<th>Image size</th>
<th>Latency [clock cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global read</td>
</tr>
<tr>
<td>$N = 384 \times 280$</td>
<td>385</td>
</tr>
<tr>
<td>$N = 640 \times 480$</td>
<td>641</td>
</tr>
<tr>
<td>$N = 1280 \times 720$</td>
<td>1281</td>
</tr>
</tbody>
</table>

Table 7.3 shows how many clock cycles each pipelined loop in the CDF kernel takes.

Table 7.4: Clock cycle cost of Interpolation kernel.

<table>
<thead>
<tr>
<th>Image size</th>
<th>Latency [clock cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pipe read</td>
</tr>
<tr>
<td>$N = 384 \times 280$</td>
<td>12297</td>
</tr>
<tr>
<td>$N = 640 \times 480$</td>
<td>20487</td>
</tr>
<tr>
<td>$N = 1280 \times 720$</td>
<td>40968</td>
</tr>
</tbody>
</table>

Similar to Table 7.3, Table 7.4 shows how many clock cycles each pipelined loop in the Interpolation kernel takes.
7.1.2 Area Utilization

The area of the design is measured in amount of resources used. Available resources are Block Random Access Memories (BRAM), Digital Signal Processing Blocks (DSP), Flip-flops (FF) and Look Up Tables (LUT). Data in this section comes from the HLS estimation report provided by SDAccel.

*Table 7.5: Resource usage for each kernel. $N = 384 \times 280$.*

<table>
<thead>
<tr>
<th>Kernel</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF</td>
<td>54 (2.5%)</td>
<td>1 (0.0004%)</td>
<td>12650 (1.9%)</td>
<td>17718 (5.3%)</td>
</tr>
<tr>
<td>Interpolation</td>
<td>144 (6.7%)</td>
<td>67 (2.4%)</td>
<td>21828 (3.3%)</td>
<td>24390 (7.4%)</td>
</tr>
</tbody>
</table>

*Table 7.6: Resource usage for each kernel. $N = 640 \times 480$.*

<table>
<thead>
<tr>
<th>Kernel</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF</td>
<td>70 (3.2%)</td>
<td>1 (0.0004%)</td>
<td>16577 (2.5%)</td>
<td>28233 (8.5%)</td>
</tr>
<tr>
<td>Interpolation</td>
<td>310 (14.4%)</td>
<td>87 (3.2%)</td>
<td>36432 (5.5%)</td>
<td>41273 (12.4%)</td>
</tr>
</tbody>
</table>

*Table 7.7: Resource usage for each kernel. $N = 1280 \times 720$.*

<table>
<thead>
<tr>
<th>Kernel</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF</td>
<td>110 (5.1%)</td>
<td>1 (0.0004%)</td>
<td>29781 (4.5%)</td>
<td>29085 (8.8%)</td>
</tr>
<tr>
<td>Interpolation</td>
<td>650 (30.1%)</td>
<td>167 (6.1%)</td>
<td>87705 (13.2%)</td>
<td>96815 (29.2%)</td>
</tr>
</tbody>
</table>
7.2 RDC

This section contains the test results for the radial distortion correction implementations.

7.2.1 Latency

First the overall frame latency is presented. RDC consists of only one kernel and the performance of the major loops are shown in order to verify what function is the most time consuming.

**Table 7.8:** Frame latency for the different implementations of RDC using CPU as OpenCL platform.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Latency [ms]</th>
<th>N = 1280 × 720</th>
<th>N = 1600 × 900</th>
<th>N = 1920 × 1080</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU model</td>
<td>48.98</td>
<td>77.22</td>
<td>109.50</td>
<td></td>
</tr>
<tr>
<td>CPU port</td>
<td>1346</td>
<td>2078</td>
<td>2832</td>
<td></td>
</tr>
<tr>
<td>FPGA (double)</td>
<td>14.83</td>
<td>22.79</td>
<td>31.89</td>
<td></td>
</tr>
</tbody>
</table>

CPU port is the CPU model with minor adjustments to allow execution on the FPGA. FPGA is the fastest of the implementations described in chapter 6, i.e. the double cache implementation.

**Table 7.9:** Frame latency for the different implementations of RDC using FPGA as OpenCL platform.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Latency [ms]</th>
<th>N = 1280 × 720</th>
<th>N = 1600 × 900</th>
<th>N = 1920 × 1080</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU port</td>
<td>12216</td>
<td>17770</td>
<td>24250</td>
<td></td>
</tr>
<tr>
<td>FPGA (double)</td>
<td>4.07</td>
<td>6.72</td>
<td>9.98</td>
<td></td>
</tr>
</tbody>
</table>

**Table 7.10:** Burst loop initiation interval and iteration latency for different cache configurations.

<table>
<thead>
<tr>
<th>Partition factor</th>
<th>Clock cycles [cc]</th>
<th>Initiation interval</th>
<th>Iteration latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Initiation interval is the time between the start of the iterations of the loop, and the iteration latency is how long an iteration is. The values in Table 7.10 is valid for all three implementations as they all use the same burst read method.
Table 7.11: Correction loop behaviour for the different implementations.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Clock cycles [cc]</th>
<th>Trip count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Initiation interval</td>
<td>Iteration latency</td>
</tr>
<tr>
<td>Single cache</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>Double cache</td>
<td>1</td>
<td>39</td>
</tr>
<tr>
<td>NN interpolation</td>
<td>1</td>
<td>35</td>
</tr>
</tbody>
</table>

In Table 7.11 a cache partition factor of 32 is used. Trip count is the total amount of iterations.

7.2.2 Area Utilization

The area of the design is measured in amount of resources used. Available resources are Block Random Access Memories (BRAM), Digital Signal Processing Blocks (DSP), Flip-flops (FF) and Look Up Tables (LUT). Data in this section comes from the HLS estimation report provided by SDAccel.

Table 7.12: Resource usage for different implementations. \(N = 1280 \times 720\).

<table>
<thead>
<tr>
<th>Implementation</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU port</td>
<td>76 (3.5%)</td>
<td>67 (2.4%)</td>
<td>34671 (5.2%)</td>
<td>76586 (23.1%)</td>
</tr>
<tr>
<td>Single cache</td>
<td>316 (14.6%)</td>
<td>25 (0.9%)</td>
<td>6792 (1.0%)</td>
<td>9541 (2.9%)</td>
</tr>
<tr>
<td>Double cache</td>
<td>444 (20.5%)</td>
<td>25 (0.9%)</td>
<td>7170 (1.1%)</td>
<td>10476 (3.2%)</td>
</tr>
<tr>
<td>NN interpolation</td>
<td>316 (14.6%)</td>
<td>34 (1.2%)</td>
<td>6960 (1.0%)</td>
<td>9965 (3.0%)</td>
</tr>
</tbody>
</table>

Table 7.13 displays the influence of cache partitioning on FPGA resources.
Table 7.14: Area utilization for single cache implementation depending on image size.

<table>
<thead>
<tr>
<th>Image size</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = 1280 \times 720$</td>
<td>316 (14.6%)</td>
<td>25 (0.9%)</td>
<td>6792 (1.0%)</td>
<td>9541 (2.9%)</td>
</tr>
<tr>
<td>$N = 1600 \times 900$</td>
<td>380 (17.6%)</td>
<td>25 (0.9%)</td>
<td>6718 (1.0%)</td>
<td>10213 (3.1%)</td>
</tr>
<tr>
<td>$N = 1920 \times 1080$</td>
<td>476 (22.0%)</td>
<td>25 (0.9%)</td>
<td>6712 (1.0%)</td>
<td>10370 (3.1%)</td>
</tr>
</tbody>
</table>
This chapter discusses the systems implemented and the results that were achieved.

8.1 Implementation

This section will discuss the implementation in general and evaluate the programming tools available in SDAccel.

8.1.1 Design Choices

The idea for the CLAHE implementation were originally to have a single kernel similar to the RDC implementation. This was soon understood to be infeasible as the compiler were unable to perform HLS for even the smallest image sizes. The major problem was the local cache structure which had to be able to allow both burst write and parallel pixel reads. Even though the burst write and parallel read would never occur simultaneously the compiler was not able to determine the access patterns. At a point it was obvious that the compiler preferred smaller kernels and as a consequence it made it possible to try out the OpenCL pipe structure. Even using the final design the larger image sizes were still not synthesizable. The main reason for this may be the increasing number of blocks the were to be processed in parallel (in theory). Due to the block size of 64 × 8 the number of blocks in parallel would be 20, 25 and 30 for N = 1280 × 720, N = 1600 × 900 and 1920 × 1080 respectively. As can be seen in Table 7.4 the largest image size achieved was N = 1280 × 720 with total latency of 75057 clock cycles for the processing of one row of blocks.

As the algorithm is implemented so that it processes an row of blocks at a time, the resulting latency can be approximated with the combined iteration latency of the two kernels, i.e. 121442 clock cycles for N = 1280 × 720. This is however the
worst case performance as the two kernels can work somewhat simultaneously thanks to the FIFO register (OpenCL pipe). Though, as the interpolation kernel is slower the CDF kernel will most likely be stalled until the interpolation kernel has emptied the FIFO.

The RDC kernel is processing all pixels in a serial manner as the calculations are quite simple, for \( N = 1920 \times 1080 \), would require around \( 2 \times 10^6 \) clock cycles for an entire image. As can be seen in Table 7.11 this was not possible using the single cache configuration. The FPGA does contain true dual port BRAM blocks which implies that an initiation interval of 1 should be possible thanks to the array partitioning storing adjacent pixel columns in different modules. An explanation may be that the compiler still tries to reduce the resource usage and packs pixels into the BRAM even though explicitly instructed to partition the array in a specific manner. The BRAM elements have size of 32 bits, allowing four pixels to be stored in the same cell. Whether this is case is not known.

### 8.1.2 Latency

In Table 7.3 the clock cycle cost for one main loop iteration of the CDF kernel is presented. The stages Calculate histogram, Clip and CDF are the three loops that in theory should be processed in parallel for each block. Expected clock cycle costs for the loops are 512, 256 and 256 respectively as the first is dependent on block size and the other two are dependent on number of bins in the histogram. As can be seen only Clip takes the expected amount of cycles to complete. The CDF loop is consistent for all image sizes but still takes twice the amount of cycles. This behaviour could originally be explained with the fact that the loop accesses two elements from the CDF array in each iteration. This problem was handled by using temporary variables in an attempt to emulate the pipeline forwarding. However, the compiler was still not able to realize the pipelined loop with an initiation latency of one.

The Calculate histogram loop should have an iteration latency of 512 clock cycles, as it must iterate over all pixels in a block, which is not the case as can be seen in Table 7.3. It is also not consistent for different image sizes. As the pixel memory is partitioned so that each block is stored in a separate memory this shows that the compiler is not able to determine that there will be no memory port collisions.

For the interpolation kernel the loop of interest is the Interpolate loop, the cycle cost is presented in Table 7.4. The Interpolate loop should have a clock cycle cost equal to the block size, i.e. 512, due to the fact that each pixel is interpolated. This is not the case. Similar to the Calculate histogram loop, the Interpolate loop is also inconsistent for different image sizes, again showing that the compiler cannot determine BRAM access patterns.

The performance of the RDC was more of what could have been expected from the type of implementation. However, as the FPGA has clock frequency of 250 MHz, which implies an execution time of 8ms for \( N = 1920 \times 1080 \), it can be stated that the burst read/write to the global memory is a time consuming task that can not be overlooked. In Table 7.10 the burst cache is partitioned in order
8.1 Implementation

to maximize performance.

8.1.3 Area Utilization

For the CLAHE implementation the most unexpected was the under-utilization of DSP-blocks. As can be seen in Table 7.5 to 7.7, only a single DSP block is used in the CDF kernel. Going back to the design one can argue that a DSP block could have been useful when calculating a blocks actual CDF. That would result in at least one DSP per block, i.e. 6, 10, 20 for $N = 384 \times 280$, $N = 640 \times 480$ and $N = 1280 \times 720$ respectively. A theory that can explain the low usage may be that the DSP blocks are reserved for operations including multiplications. As the CDF kernel mainly involves additions, which are less time consuming, the functions are instead mapped to LUTs. The interpolation kernel uses more DSPs, and it can be as a consequence of performing more multiplication during the interpolation. Seen in Table 7.12 the CPU port is using much more DSPs, FFs and LUTs than the FPGA optimized versions, but less BRAMs. The reason behind is the pixel caches in the FPGA versions. The CPU port is using BRAMs for temporary variable storage which can be a consequence of the very serial algorithm.
8.2 Exploiting Parallelism with SDAccel

As can be seen in both Table 7.2 and Table 7.9 the massive parallelism available on FPGAs is poorly utilized in the case of direct porting of the code. This section discusses the methods used to increase the performance and better utilize the resources.

8.2.1 Burst Memory Access

The memory controller of the global memory pack 16 uint16s together to maximize the bandwidth, resulting in a final package size of 1024 bytes. A burst access is then applied which serially transfers these large packages, skipping the overhead in between each transfer. To achieve a burst both implementations use a local cache (on-chip memory i.e. BRAM) which temporarily stores a chunk of the image. A conclusion that can be made is that the optimal size for the cache to minimize the transfer overhead would be to store the complete image. This is in practice not very desired for most cases as it consumes much of the FPGAs resources. For the KU3 board each BRAM is 18kbit large, and for an image with a resolution of 1920x1080 pixels it is in theory possible to store all of pixels. It is however not recommended as the OpenCL structure requires resources for axi interface.

8.2.2 Loop Unrolling

As seen in Tab. 7.11 unrolling the main loop gives a great performance boost for the NN interpolation version of RDC. This can be explained from the implementation. Each iteration first calculates the distorted pixel coordinates, then reads between one to four pixels from the input cache and lastly writes one pixel to the output cache. At first this seems to easily unrollable but the problems arise at the input cache read. There is a possibility for a pixel to have it distorted coordinate match up with neighbouring pixels, resulting in multiple reads from the same BRAM address. As mentioned, each BRAM has two access ports which means that two pixels can be retrieved simultaneously, but no more than that. Closer the center of the picture there are risk of many adjacent pixel having roughly the same distorted coordinate.

8.2.3 Data Types

SDAccel supports a series of data types from the OpenCL standard. Exceptions are 64 bit types such as double and uint64_t. During the optimization of the radial distortion correction algorithm for FPGA the compiler was experiencing difficulties when trying to pipeline a certain loop in the program. The solution was to replace all floating point operations with fixed point operations. This may seem as an obvious path to follow as the FPGA has no floating point units (FPUs). However, while inspecting the high level synthesis log during the compilation of the original program one could see that the compiler automatically inserts an
float-to-uint32_t function for each floating point variable. Examining the data sheet for Vivado HLS it can be noted that the HLS simplifies the floating point calculation as much as possible but does not perform any optimization. Optimizations in this sense are to use the FPGA resources more efficient to gain extra precision during the fixed point calculations.

One possible reason to why the compiler were not able to pipeline the loop can be that the float-to-uint32_t function, in it’s most area efficient implementation, is an iterative process when shifting down the exponent to retrieve the correct fixed point value. As the number of floating point operations increase so does the amount of pipeline stages required, in this case to an infeasible point. The implementation for the radial distortion correction does after the optimization instead use fixed point system realized using uint32_t with 16 bits for the fractional part. This precision were considered good enough for the calculations but as there were many multiplications the partial results had to be shifted to remain inside the range of 32 bits, utilizing only 8 bits for the fractional part. This could have been partially avoided if one could make use of the 64 bit integers or arbitrary precision data type available in Vivado HLS.
8.3 OpenCL

The main argument for using the OpenCL environment was the portability of code between various platforms. As can be seen in chapter 7 this was indeed possible for the functional aspect, but not for the performance. As can be seen in Table 7.9 a straight port of the CPU implementation resulted in surprisingly low performance on the FPGA. After major modifications to the implementation the algorithms was able to run smoothly on the FPGA. Many optimizations that were performed relied on the underlying structure of the FPGA (BRAMs, LUTS etc.) which means that a software developer with little or no FPGA programming experience would find it difficult achieve the desired performance boost.

8.3.1 SDAccel

The SDAccel IDE provides information about possible bottlenecks and presents solutions to the specific property. This gives the programmer a quick hint of what can be improved but as the examples are very simplified they can fail to cover the more complex case for your code. The flow of the development is however smoother as many problems are detected early. The resulting performance of the implementations was however lower than expected.

Compiler

Overall, the major limiting factor was the compiler in SDx 2016.4, which turned out to be underwhelming in terms of optimization. To reveal parallelism the code had to be written in a simplified manner. One example of this was the multiple layered loops. To allow an inner loop to be unrolled the body had to be small and contain as few variables as possible. When accessing arrays a maximum of one variable was often the case. Using more variables, even though there were no possibility of collision when reading from or writing to the array, the compiler could not sort out the dependencies and had to stall outer pipelined loop to maintain functional correctness.
8.4 Node Placement

As a consequence of the OpenCL computing system visualized in Figure 3.1 it can be expected that the Image Processing Node (IPN) will not be preferred for case A and C in Figure 2.2. This due to the data transfer in the OpenCL framework that infers a latency which may too large.

As can be seen in Figure 4.2 the KU3 FPGA board contains both SATA and SFP ports. If those interfaces can be used the unnecessary data path via the CPU can skipped and data can be directly accessed in the global memory on the FPGA. As of today the interfaces are not possible to use with the SDaccel API directly. However, they can be used if the structure of the axi control unit on the FPGA is modified. This must be done in HDL and most probably requires time to learn.

According to Xilinx, it will in the future be possible to directly feed the data into the accelerator, atleast via the SFP port, with a simplified interface in the SDaccel API. If such a feature were to be implemented the framework would still come short as it requires an host processor for controlling the accelerator.

For VDS this does open up the possibility to implement the IPN inside at position C as there exists modified display nodes today containing a CPU.
This chapter concludes this thesis. It summarizes the answers to the research questions and brings up what can be done in the future for this topic.

9.1 Research Questions

• Are the image processing algorithms suitable for use where low latency is a critical factor?
  Yes, but it depends on the exact requirement on latency. Both CLAHE and RDC has data dependencies which can cause problems. For CLAHE the block effects can still be visible using a single step of interpolation. The effect is only visible in areas containing homogeneous light levels. As mentioned in chapter 5 this can be solved by additional interpolation which further increases the latency. However, this induces latency which will be one to two rows of blocks which is negligible using the block size and image resolution presented in this thesis.

  For RDC, the lens is determining the latency induced by the dependencies. As mentioned in chapter 5 the lens used induces a dependency equal to 120 rows of pixels. This lens was not considered wide angle but the amount of rows was still quite high. For a real wide angle lens latency may be a problems that hinders the use of the algorithm.

• Is OpenCL a suitable choice of framework for implementing the algorithms?
  No, or atleast not by using SDAccel 2016.4. As can be seen in chapter 7, the frame latency is too long for both algorithms when the resolution is increased. For lower resolutions SDAccel creates a system that is usable but nothing that is suitable for future systems with demands on higher reso-
lution. For the case of CLAHE where the targeted resolution could not be synthesized this demonstrates that HLS is still not a viable option compared to classic FPGA development.

9.2 Future Work

As this thesis only touches a fraction of what is possible using HLS there is more that can be done in the future. This section brings up some aspects of interest.

9.2.1 Optimization

As the time were limited the algorithms are not ideally optimized. Parameters that was not fully explored was the data flow or work item pipelining directives. By looking through the compilation log it could be seen that the data flow directive was somewhat used in the design through automatic insertion by the compiler. How this affected the performance for our algorithm is not known but by looking at the examples provided by Xilinx some performance can be gained.

Except for the use of available programming tools the overall structure of the implementations could have been altered to increase the performance even further. One requirement for the use of OpenCL in node A, see Chapter 2, would be to have a continuous data stream into the system. This would completely remove the need for burst transfers and FIFOs might have been a better choice for local pixel storage, compared to a cache. Nevertheless, as this approach is still not available in SDAccel it will need to be investigated in the future.

Another aspect that is interesting is to optimize the implementations for lower resource usage or power consumption.

9.2.2 New Software Tools

The compiler in SDx 2017.1 has major improvements compared to the compiler in 2016.4. Some examples of this is analysis of memory access patterns to better utilize memory access coalescing. If implemented correctly, this may solve the problems encountered. It may also deem the burst cache unnecessary and be able to hide memory latency in manner similar to GPUs.


