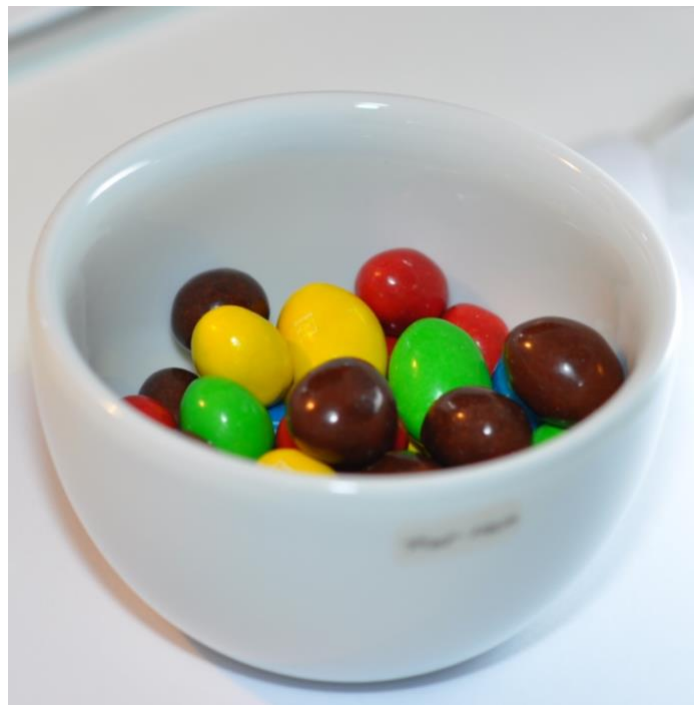


# Accelerated K-Nearest Neighbours Colour Classification System

Machine Learning hardware implementation using MiniZed, a Xilinx Zynq-7000 SoC

Advanced Digital Design 2020 (E5ADD)

Aarhus University, School of Engineering  
Campus Herning



Advisor: Morten Opprud Jakobsen (morten@ase.au.dk)

Handed in on 18<sup>th</sup> December 2020

Authors:

**Steffen Præstholt Breinbjerg**

Student id: 201700288

steffenpb@outlook.dk

**Janus Bo Andersen**

Student id: JA67494

janus@janusboandersen.dk

## Summary

In this project, we implement a hardware-accelerated k-nearest neighbour classifier on a small Xilinx Zynq-7000 device, the Z7007S AP SoC on the MiniZed board from Avnet.

We first review the design methodology, and then do an exploration of modern Xilinx tools available to implement hardware-accelerated machine-learning algorithms on this device. It turns out that for the MiniZed, the most feasible approach is to use the 2019.1 toolchain to get the most rapid development flow.

Next, we analyse the k-nearest neighbours classifier (KNN) and its acceleration potential. It is a relatively simple, but in practice very powerful, classifier that is often used in real-world applications. We also develop a dataset of M&Ms colours, which is used for testing our implementation of the KNN.

To develop the system, we start with a base board file developed in Vivado 2019.1, and an envisioned conceptual design for the system. Using SDSoc and HLS, we then implement the classifier, and hardware-accelerate the most compute-intensive part of its algorithm. The classifier is partitioned so the parallelizable work is in PL, and the sorting/search part of the algorithm, along with a small interface giving the user a choice of the hyperparameter  $K$ , is run in the PS. The algorithm is verified to perform correctly versus a benchmark implementation in Matlab.

The final system is based on a design space exploration. During this process, we refine an accelerator design that achieves 7.4x speed versus a similar implementation on the PS, for a problem size of classifying a 64-dimensional vector in a space of 1024 64-dimensional reference vectors. Considering that the accelerator runs at 100 MHz in the PL, and the Arm Cortex-A9 runs at 667 MHz with SIMD instructions in the PS, it is a very significant gain.

When running the final tests on the M&Ms dataset, the final performance gain is around 6.9x.

In all, it turns out that there is a high number of tuneable factors in achieving meaningful speed-ups, among which are clock frequency of the accelerator, efficient design of data motion between PS and PL, efficient algorithmic design, loop, pipeline and dataflow optimizations and in general structuring the size of the problem.

Further work remains to explore the design-space fully.

## Table of Contents

<b>1</b>	<b><i>Introduction .....</i></b>	<b><i>1</i></b>
<b>2</b>	<b><i>Problem statement and requirements.....</i></b>	<b><i>2</i></b>
<b>3</b>	<b><i>Methodology .....</i></b>	<b><i>4</i></b>
<b>4</b>	<b><i>Toolchain exploration.....</i></b>	<b><i>6</i></b>
<b>5</b>	<b><i>Development flow.....</i></b>	<b><i>11</i></b>
<b>6</b>	<b><i>Analysis .....</i></b>	<b><i>12</i></b>
<b>7</b>	<b><i>Design.....</i></b>	<b><i>16</i></b>
<b>8</b>	<b><i>Test, deployment and final implementation.....</i></b>	<b><i>27</i></b>
<b>9</b>	<b><i>Verification .....</i></b>	<b><i>30</i></b>
<b>10</b>	<b><i>Discussion .....</i></b>	<b><i>31</i></b>
<b>11</b>	<b><i>Conclusion .....</i></b>	<b><i>31</i></b>
<b>12</b>	<b><i>Improvements and future work.....</i></b>	<b><i>31</i></b>
<b>13</b>	<b><i>Terminology .....</i></b>	<b><i>32</i></b>
<b>14</b>	<b><i>References.....</i></b>	<b><i>35</i></b>
<b>15</b>	<b><i>Appendices.....</i></b>	<b><i>36</i></b>

# 1 Introduction

The purpose of the project is to gain practical problem-solving experience with hardware / software co-design using the **Zynq-7000 AP SoC** and Xilinx development tools. This SoC system combines a single-core Arm Cortex-A9 processor and Xilinx Artix-7 programmable logic (FPGA fabric).

The team has decided to focus on implementation of machine learning (ML). This is an area ideal for hardware acceleration due to the often highly parallelisable nature of ML algorithms, the typical requirements for high data throughput, and because ML algorithms are often deployed to small edge devices for use-cases that require real-time performance.

The specific focus is to develop a prototype **colour recognition system** using the **k-nearest neighbours** classifier. There is however nothing special about colours, as the colour of a pixel (or a neighbourhood of pixels) can be represented by a feature vector like any other typical classification problem, such as recognizing hand-written digits. So, the implemented solutions will have **general applicability**.

In normal practice, a dimensionality reduction technique such as Singular Value Decomposition (SVD) or Principal Components Analysis (PCA) is applied beforehand to bring the feature space to a manageable size, e.g. from millions of pixels in an image to 50-200 more meaningful features. Another effect is to reduce noise. When a set of feature vectors are transformed into a lower-dimensional feature space, the direct connection to what was observed is often lost anyway<sup>1</sup>.

Nonetheless, colour recognition is a problem with relevance in industry, e.g. production (quality control, sorting machines), automotive (environment inference, self-driving cars), robotics (colour-based object tracking), and many other application areas. It is a problem of sufficient complexity such that it can be meaningfully accelerated using hardware, yet not too complex to detract from the main purpose of the project.

The project is part of the elective course “Advanced Digital Design” (E5ADD) on the 5<sup>th</sup> semester of the electronic engineering degree at Aarhus University, School of Engineering, Campus Herning. The advisor was associate professor Morten Opprud Jakobsen.

Abbreviations and acronyms used in this report are explained in the section preceding the references. References to literature and guides are stated using IEEE referencing.

---

<sup>1</sup> This transforms the dimensionality of the actual samples into a feature space with reduced dimension. E.g. sampling image points as RGB-triplets with 8-connected neighbourhoods gives 27-dimensional feature vectors, as  $(1+8)*3=27$ . Reducing the dimensionality might yield a 9-dimensional feature space that adequately describes the variance of the data set. The 9 “new” features are combinations of the 27 “old” features, but it is not a priori clear how the new features will be constructed.

## 2 Problem statement and requirements

The problem is to co-design and implement part of a vision system to discriminate between different colours of M&Ms. The system-to-be must be implemented on the Zynq-7000 SoC.

### 2.1 Rich picture

The Rich Picture shows the envisioned application of the system-to-be (from original project pitch). The system-to-be is limited to the Zynq-7000-based colour classification system.

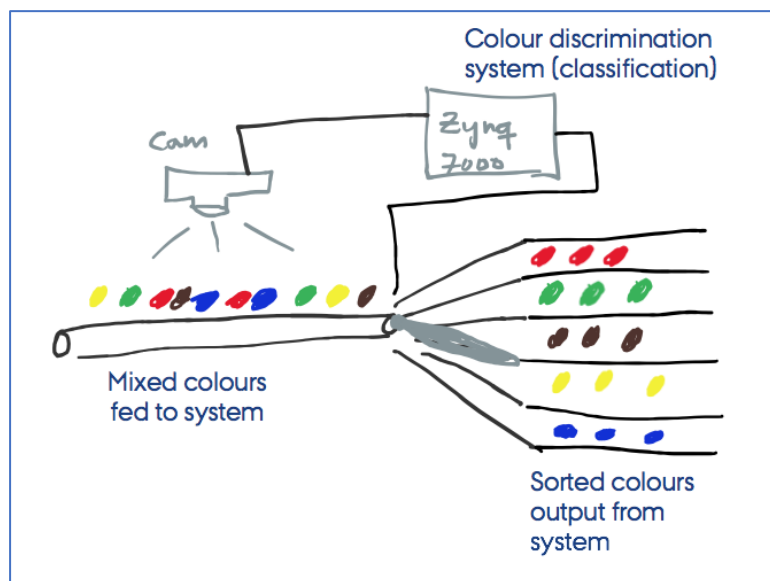


Figure 1: The system-to-be implements part of a vision system that can recognize and sort objects of different colours, in this case M&Ms. The system-to-be on the Zynq-7000 SoC is responsible for discriminating between the different colours.

### 2.2 Requirements

The system requirements are derived from the original project proposal and refined.

Table 1: Requirements specification and verification tests for the system-to-be

Req.	Requirement description	Verification test
1	The system must be able to separate more than two colours from the M&Ms colour palette.	Run a test data set containing three colours on the system. Algorithm must have same accuracy as benchmark KNN algorithm in Matlab using same data.
1-A	<i>Nice-to-have improvement:</i> All 5 colours in the M&Ms colour palette can be classified.	Same with all 5 colours.
2	The system must take as input sampled image patches of RGB colour. The input is structured as N-dimensional feature vectors. The system must take R	Generate a dataset with 1024 reference vectors and 50-60 features describing RGB samples in an image patch. Test the dataset using a benchmark algorithm and

	reference vectors (training data) and Q query vectors (to be classified), with $R \gg Q$ .	require the same performance from the Zynq system.
3	Input is stored on the eMMC	No test.
4	The system must output the recognised colour as text or code to a terminal.	Reverse functional test. The system outputs the errors when classifying labelled data. The errors must be human-readable in a terminal.

### 2.3 Delimitations

The project must be within the 5 ECTS course limit, which only allows a relatively narrow scope. So, the project must be delimited, and a number of interesting aspects will not be considered due to scope/time constraints.

First, the acquisition problem (how to get images or pixel brightness values into the SoC in the first place) will not be considered. This would involve expanding the hardware platform with a camera. The acquisition problem is furthermore not *directly* considered central to the focus of the problem of *acceleration* using an SoC.

Second, and in relation to acquisition, we do not focus on image pre-processing on the SoC platform. So, the problem of algorithmically obtaining classifiable feature vectors from images is considered outside scope. This could involve de-noising, segmentation/region selection, and colour space transformations<sup>2</sup>.

Third, we do make some attempts to tweak algorithms for higher efficiency, but mainly in terms of achieving fast interfaces to PS and leveraging SDS/HLS. We delimit the project from experimenting with the many more clever and efficient algorithms that e.g. partition matrices into multiple sub-blocks to achieve higher parallelism in derived hardware.

Fourth, we do not make VHDL testbenches to verify and validate the individual hardware blocks. The hardware is derived from C++ by HLS, so we test and verify from a software system level, meaning that we confirm that the entire accelerator flow gives correct results.

---

<sup>2</sup> In image processing it is not uncommon to use a different colour space than RGB, because RGB is sensitive to overall luminance (brightness) in a nonlinear way. I.e., illuminating an object at different luminance levels will affect the ratios between R, G and B values. This obviously makes it hard to use RGB values for classification in an environment where lighting is not stable. Transformation of RGB into the YCbCr colour space yields gives a specific luminance component (Y) and two colour/chroma components, red-difference and blue-difference. The latter are (more) stable for metrics and classification.

### 3 Methodology

#### 3.1 Co-design method

We apply the double-roof model of co-design, which in short is a process of allocation, binding and scheduling [1]. It begins from the system level requirements, and branches out separate software and hardware design chains. Following this method, the developer successively refines the system synthesis by mapping functional/behavioural specs. to structural implementations.

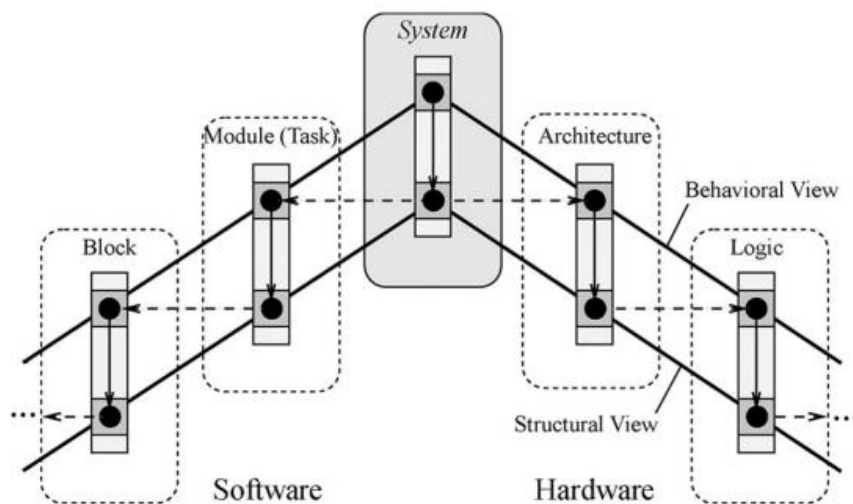


Figure 2: The double-roof model of co-design is stepwise with successive hardware and software refinement steps [1].

The specific mapping tasks in the design process are detailed in Table 2.

Table 2: The three specific mapping steps in co-designing an embedded system.

Action	Involves	Specifically for this project
Allocation	Composing architecture. I.e., selecting system resources, IP blocks, libraries, interfaces and interconnections.	<ul style="list-style-type: none"> <li>Designing interfaces between blocks (see design chapter)</li> <li>Selection of data movers and storage areas.</li> </ul>
Binding	Mapping required functionality to processing resources.	<ul style="list-style-type: none"> <li>Partitioning the functionality onto the PS (UI and sort/search) and PL (compute-intensive acceleration).</li> <li>Binding MAC operations to DSP blocks in the PL.</li> </ul>
Scheduling	Selecting timing for when functionality runs on a resource, order of execution.	<ul style="list-style-type: none"> <li>Forcing concurrency using various directives in SDS/HLS</li> <li>Designing data flow to/from accelerator and loop timing inside accelerator.</li> </ul>

### 3.2 High-level synthesis and SDSoC/Vitis method

In general, HLS is a design method for synthesizing high-level languages, e.g. C/C++, to RTL-level hardware descriptions, such as VHDL. It supports HW design from a higher level of abstraction than available with HDLs, allowing the developer to focus on behavioural and algorithmic aspects of the design, and to perform rapid design space exploration. Xilinx offers different tools that support this design method, among others SDSoC and Vitis<sup>3</sup>.

SDSoC and Vitis are tools and methods to manage resources and perform HLS as a co-design process. That is, to design a full system partitioned across PS and PL in the SoC.

With this tool, the *“system compiler generates an application-specific system-on-chip by compiling application code written in C or C++ into hardware and software that extends a target platform”*. And the developer can *“employ the SDSoC (or Vitis) tools to customize the platform with application-specific hardware accelerators and data motion networks”* [2].

In other words, the co-design flow using SDSoC is:

- Design a base hardware platform in Vivado, including necessary PS and PL blocks.
- BSP, FSBL and OS (or just standalone/bare-metal) are generated using SDK, or via SDSoC / Vitis.
- Application code is written or imported in SDSoC / Vitis.
- On a functional level, developer identifies functions for hardware acceleration (PL).
- Accelerated functions are synthesised to HDL and integrated into HW platform.
- HW platform is synthesized and implemented into a bitstream, and the software is compiled to an ELF file. Both are packaged for deployment to the device, along with an FSBL.
- Profiling tools are used to evaluate the system.
- System refinement is iterative, the design space is explored, and the system is potentially re-partitioned.

As described, the SDSoC/Vitis flow supports the double-roof method well. In the next chapter, possible toolchains are explored further to choose one for system implementation.

---

<sup>3</sup> Other than SDSoC: Vivado HLS for HW design from C/C++/SystemC straight to HDL, SDAccel for OpenCL acceleration, and Vitis HLS (since late-2019) encompassing both direct HLS to HDL or HW/SW co-design.



## 4 Toolchain exploration

Working with Xilinx hardware/software development opens for a large toolbox, which require a lot of in-depth knowledge. Furthermore, Xilinx releases new versions of their development tools several times a year, with changes to workflow, structure etc. Additionally, a newer version might cause problems with already designed hardware or software e.g. Hardware IP's or Board Support Packets (BSP). It's essential to understand that, at some point a decision is needed on what version you are developing on. This section shows some of the work spend on exploring a different version vs. the introduced version used in the course E5ADD. The main idea is to get an understanding if it's feasible for the target platform Minized.

The course E5ADD was based on material and exercises for 2019.1 delivered by Avnet. The 2019.2 version introduced a replacement for the SDK with Vitis. For this exploration we dive into the 2020.1 version and try some of the new features relevant for this project. Specifically, we look at *Vitis AI* and the new workflow for *Vitis and Vitis HLS*.

### 4.1 Vitis HLS

A required element to work with Vitis HLS is Petalinux, which is a small embedded linux *OS* made for Xilinx processing system.

Building the petalinux requires command line tool, and either BSP or hardware design from Vivado. There is an available BSP from Avnet to 2020.1 [3]. See Figure 3. This is the full design block with Wifi, Bluetooth, GPIO, AXI and system clocks. The most important blocks for the accelerated design are system clocks and AXI-bus, everything else can be removed if not needed to save space on the PL. Furthermore, there is added an IP-block for SD-card [4] to the design (Left side of the figure).

When the hardware design is ready, the Petalinux project can be created, configured for the hardware design. See Figure 4. In order to make petalinux support acceleration and boot from SD it's necessary to configure the petalinux before building, with different boot arguments, define root filesystem type and add additional packages.

Following packages are needed:

- Xrt (Xilinx Runtime) [5]
- xrt-dev
- zocl
- opencl-headers-dev
- opencl-clhpp-dev

Finally, Linux can be built, and booted on target. If successful an SDK can be generated for Vitis environment to start developing software for the PS and accelerated design for PL.

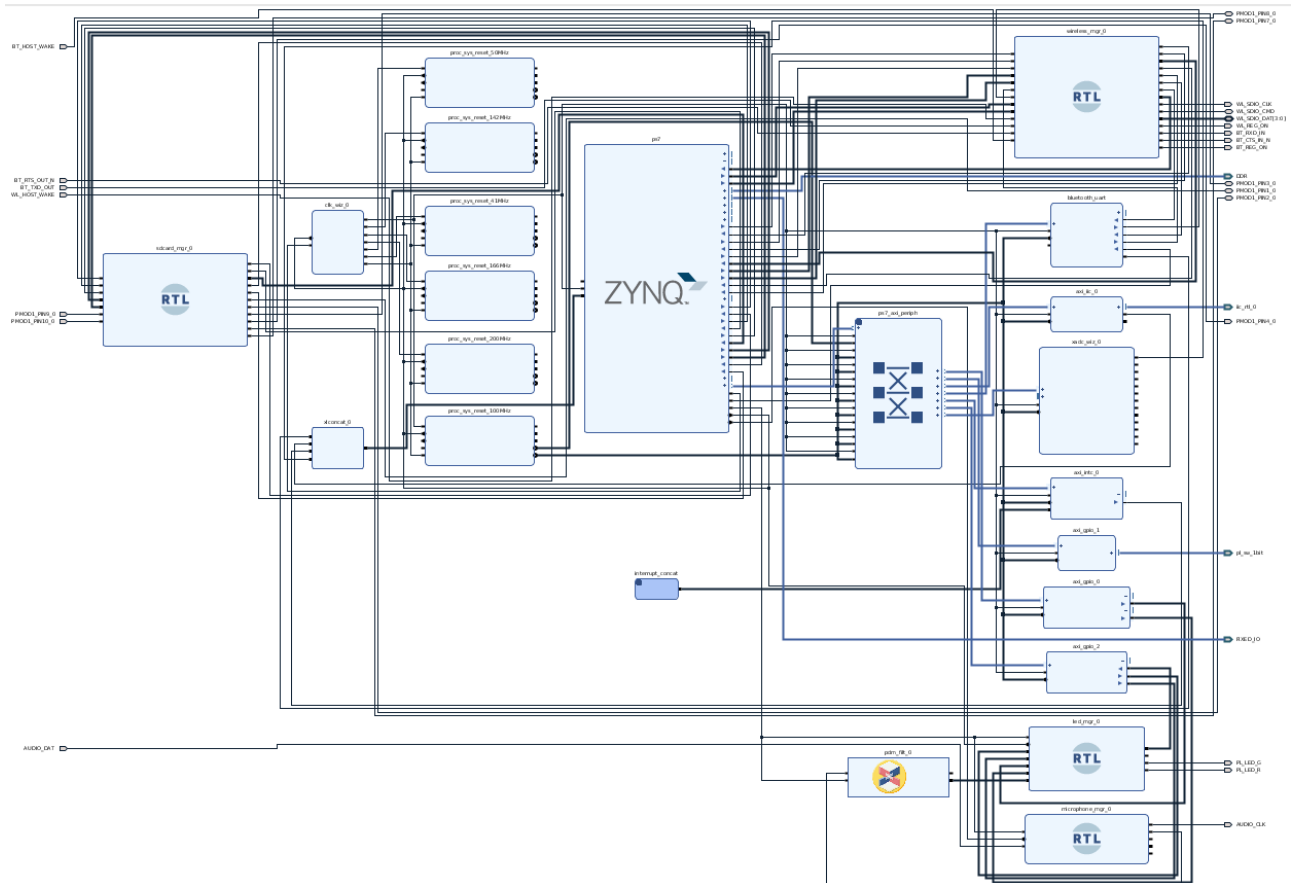


Figure 3 - Full Block diagram Petalinux

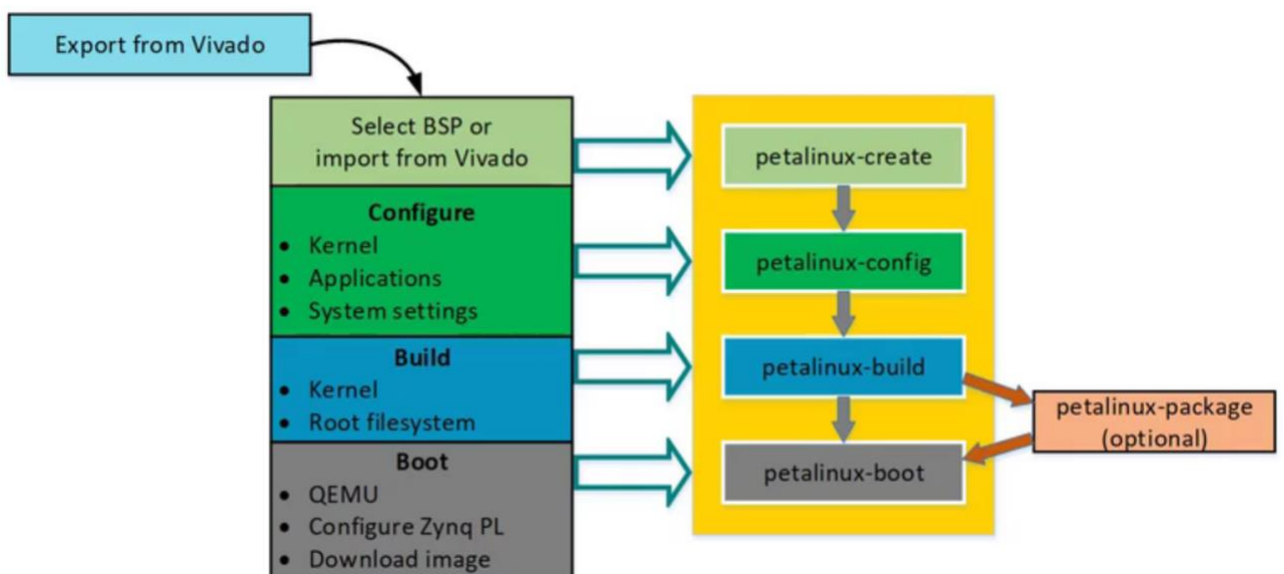


Figure 4 - Workflow for Petalinux. Right side shows commands given.

Workflow for the Vitis environment:

- Create platform project based on the Vivado design and set Linux as operating system.
- Set relevant paths for Boot files, Bit-streams and sysroot created by SDK.
- Make new application based on the Linux platform.

The new application is now ready to implement hardware with code written in C/C++. Best practice is to use the Vitis HLS tool that can generate hardware kernels for Vitis and IP-blocks for Vivado. Like previous versions of HLS, it can simulate, debug and test the code/algorithm and give detailed information about hardware use (LUTs, DSP etc) and time constraints. When built, it can be exported as Vitis Kernel (.xo) file, which can be linked by the Vitis compiler in the application acceleration flow [6].

Following the above workflow, it was possible to boot Linux on target platform but failed in enabling hardware acceleration. At this point it's hard to pinpoint where it went wrong, but something suggest it might be the petalinux-build with the required packages not loaded correctly. Given the lack of examples/guides (Vitis 2020.1), for the Minized, on this subject, and the lack of knowledge and experience makes it hard to debug. Generally, Xilinx and Avnet does provide a lot of information and guides for Vitis, but only for newer Xilinx products. With the time scope for this course in mind, the unknown amount of time required to solve the problem and implement HLS C/C++, it was decided to not go further with this.

## 4.2 Vitis AI with DPU

Vitis AI is a development environment for accelerating AI on hardware. Xilinx offer two different approaches either for edge (on target) or cloud, where focus is on *edge* for this project. Figure 5 shows the development flow overview. Vitis AI consist of IP cores, tools, libraries, models and examples designs. There is a range of already optimized models ready to use in the *Xilinx modelzoo* [7]. If a custom model is needed it's possible to build through e.g. Tensorflow.

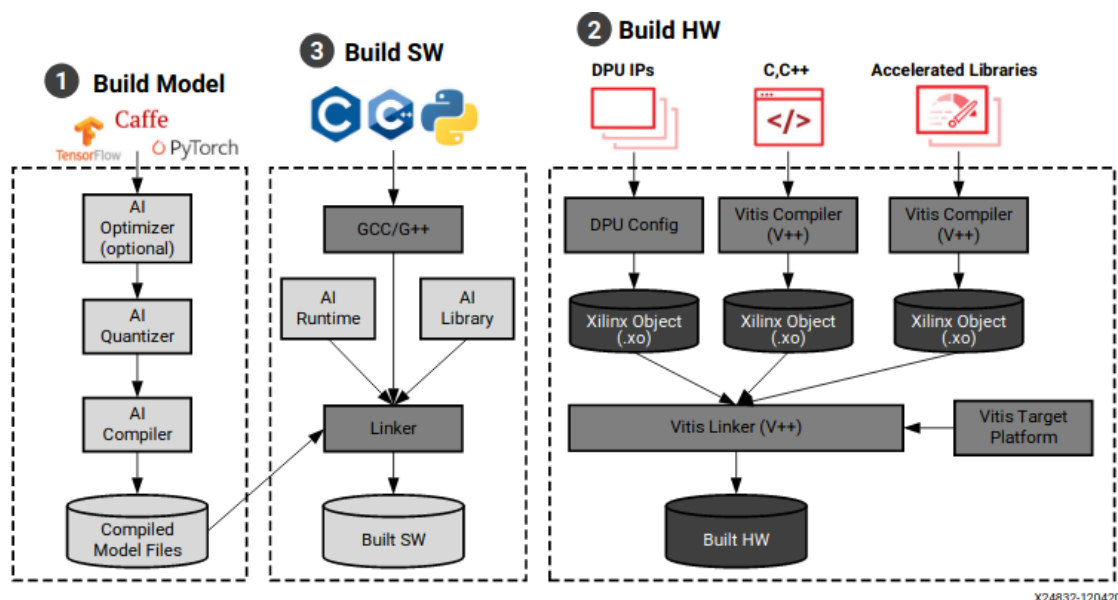


Figure 5 - Vitis AI Flow [8]

Looking into the *edge* development flow there are two tools needed: The Deep Processing Unit (DPU) and Deep Neural network Development kit (DNNDK). The DPU is an IP-Core which can be integrated into Vivado design. DNNDK [9] provide pruning, quantization, compilation, optimization, run time support for models generated by e.g. Tensorflow and C/C++ API's for handling DPU kernels.

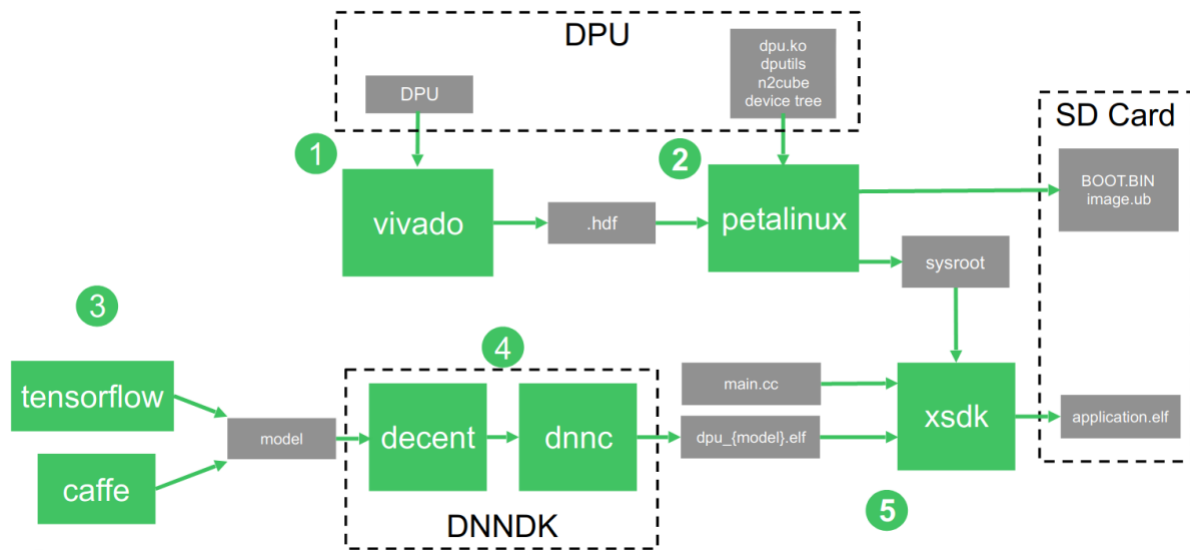


Figure 6 - Development flow Vitis AI "edge"

Figure 6 shows flow for *edge* development.

1. DPU can be integrated into custom board design. Hardware clocks, reset, and interrupts in the HW-design is necessary.
2. Final hardware design can be integrated with Petalinux. See Figure 4 for Petalinux flow. In order to make petalinux enable to use the DPU-core packages are needed in the linux build. (See DPU field to the right figure) Additionally, it's required to configure the device-tree.
3. Creating tensorflow/caffe model. This is not relevant for this project and will not be discussed further.
4. Using DNNDK to optimize and quantize the model for the target device.
5. Implement Code in Vitis using DNNDK API to manage the DPU. See Figure 7.


Routines 	
<b>dpuOpen()</b>	Open & initialize the usage of DPU device
<b>dpuClose()</b>	Close & finalize the usage of DPU device
<b>dpuLoadKernel()</b>	Load a DPU Kernel and allocate DPU memory space for its Code/Weight/Bias segments
<b>dpuDestroyKernel()</b>	Destroy a DPU Kernel and release its associated resources

Figure 7 - Example on DNNDK API Routines

The above gives a small idea on how to create an accelerated flow with Vitis AI.

As mentioned before Xilinx have many designs examples, prebuild SD-images and optimized algorithms ready to launch on different Xilinx platforms but not the Minized. Looking at design examples it's raises a concern if the Minized fabric have enough space for DPU. The smallest design found in the examples used approx. 30k LUTs where the Minized only have approx. 14K. According to Xilinx own documentation PG338 [10] it's possible to make the DPU fit on

Zynq-7000. Although with no examples and no optimized models made for the zynq-7000 series, it will require a lot of work, just to figure that the DPU might be too big to fit on the Minized fabric. With an unknown success rate to fit the DPU on target and with the time scope considered it was concluded not to go further with Vitis AI for this project.

### 4.3 Conclusion on choice of toolchain

Based on the above discussion and review of Vitis, and given that most MiniZed training materials are directed at the 2019.1 (or earlier) versions of Xilinx tools, we have decided to use the 2019.1 toolchain for the system development and implementation.

So, the toolchain for this project is:

- Vivado 2019.1 for base hardware development.
- SDSoc 2019.1 to:
  - Manage system-level resources, interfaces and handle co-design.
  - Manage Arm compiling using SDK 2019.1.
  - Manage Vivado 2019.1 HLS for cross-compiling C++ algorithms to HDL to get hardware acceleration.
  - Initiate Vivado 2019.1 for hardware synthetization and bitstream generation.
- XSCT for deployment.

This also means that Vitis cannot be used, as versions from 2019.2 and later are incompatible with projects from 2019.1 and earlier. There is a conversion feature, but in our experience it does not fully work.

We will be creating the system as a standalone, because while PetaLinux is interesting, it is not necessary for the 2019.1 development flow, and would thus more be an unnecessary complication.

## 5 Development flow

The activity flow for the system development covers the analysis and iterative design-implementation process. The flow for the development is in Figure 8.

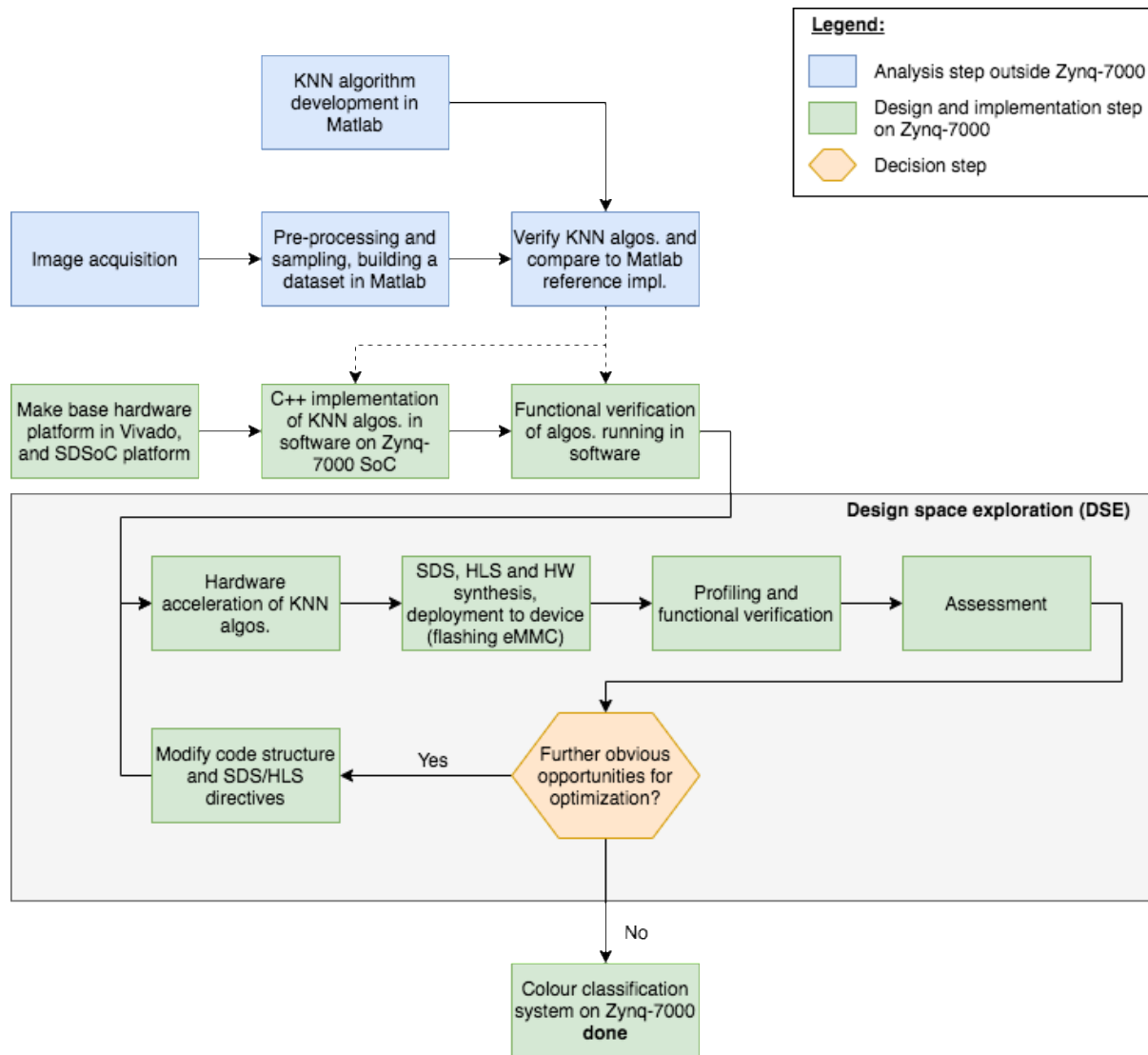


Figure 8: Analysis and iterative design-implementation process for development of the colour classification system

These “blue” steps are documented in the “Analysis” chapter. The green steps are documented in the “Design” and “Implementation” chapters.

## 6 Analysis

### 6.1 Image acquisition and M&Ms dataset

Using a DSLR, 32 images were acquired of M&Ms from a single package. The camera position was fixed, and lighting conditions were stable. Using Matlab, the images were pre-processed and labelled, resulting in the 32 cropped JPEG images in Figure 9.

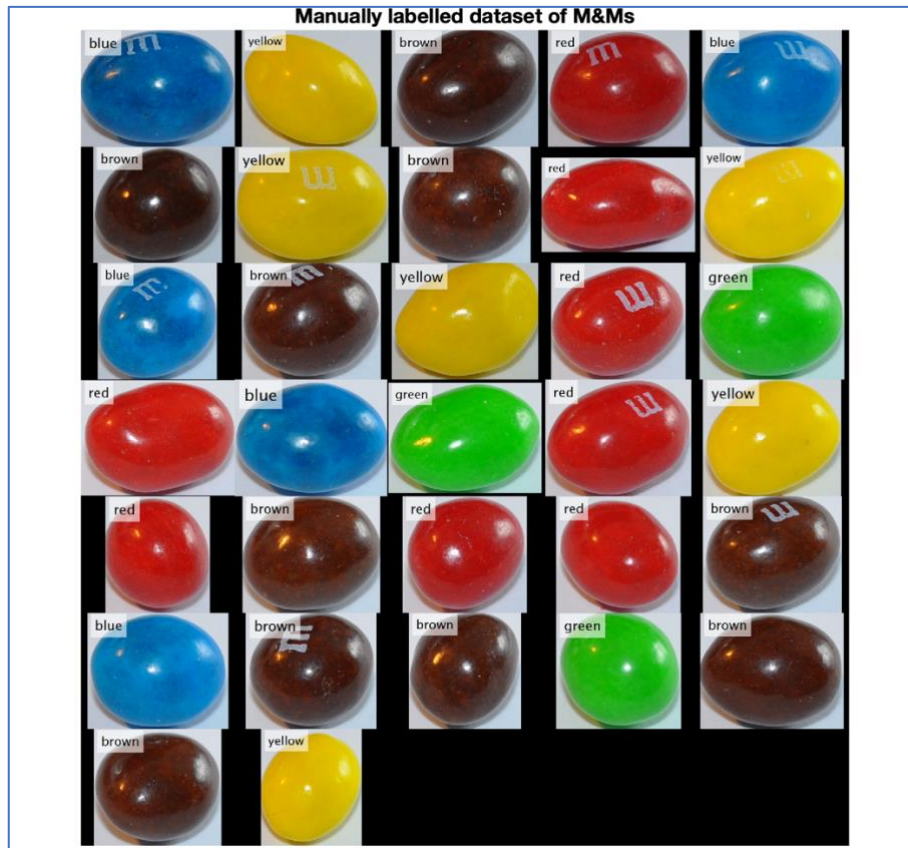


Figure 9: The M&Ms dataset contains 32 images, with samples for all of the 5 M&Ms colours.

The surface of each pictured M&M is not flat/uniform, so sampling different pixels will give different RGB values. To get a robust (realistic) KNN classifier, RGB values from 21 pixels are sampled, so feature vectors have dimension  $N = 3 \cdot 21 = 63$ . Locations are chosen randomly with a bivariate Gaussian distribution centred around the image midpoint. Changing the variance (sampling dispersion) can yield a more or less noisy dataset. An example of sampling is shown in Figure 10. In this example, 5 of 21 samples miss the M&M, introducing noise.

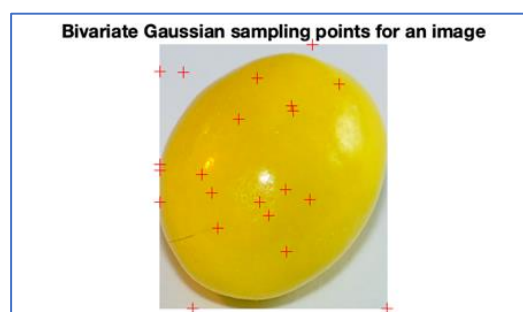


Figure 10: Sampling RGB values of 21 pixels in image 32. The red crosses mark sampled pixels.



To get a large reference set (training) and a different query set (test), each image is sampled multiple times, each time at new random pixels. In total, the dataset comprises:

- A reference set  $\mathbf{X}$  matrix of size 1024 x 63
  - 1024 reference vectors (each image sampled 32 times)
  - Each reference vector has 21 RGB samples, giving 63 integer values.
- A corresponding  $\mathbf{y}$  vector holds the 1024 labels.
- A query set  $\mathbf{X}_q$  matrix of size 100x63
  - Query vectors of the same dimension as references, but sampled at different locations
- A corresponding  $\mathbf{y}_q$  vector holds the 100 labels.

Code to generate the dataset is in appendix 15.1. Code to store the data as flat files for header files is in appendix 15.2.

## 6.2 K-Nearest Neighbour classifier

K-nearest neighbours (KNN) is a supervised learning algorithm for classification of observations into one of a number of pre-defined classes. In this project, there are 5 classes, each of the colours of the M&Ms.

It is a relatively simple algorithm to implement. It computes the distance from a query vector (test vector) to all reference vectors. The  $K$  smallest distances are found along with a list of corresponding labels. The query vector is classified as belonging to whichever class is the most frequent in the labels list. Typically, the distance metric is the Euclidian distance, or  $L_2$  norm, so for query vector  $q$  and reference vector  $r$ , the distance  $d$  is across all components as

$$d = \|q - r\| = \sqrt{(q - r)^T (q - r)} = \sqrt{\sum_{i=1}^N (q_i - r_i)^2}$$

For classification, it is not necessary to take the square root as that is monotonic function, so that will also be omitted in our implementation, and we will use  $d^2$  also just called d2.

The hyperparameter  $K$  sets how many neighbours to search for. For a general choice of  $K$ , a list of distances is sorted to find the  $K$  smallest values. If the labels are not sorted along with this list, a subsequent search of the nearest neighbour indices in the unsorted vector and a translation of the indices to labels will also be required.

An algorithm with  $K = 1$  is easier as that only requires tracking one least distance and its corresponding label. It is however also sensitive to noise, outliers and single misclassifications in the reference set.

The **main algorithm for hardware acceleration is the general KNN algorithm**. A KNN with fixed  $K = 1$  is used for comparison in some places. Both are useful in practice, and it will be interesting to understand the differences in acceleration characteristics.



### 6.2.1 Algorithm prototyping and verification

In appendix 15.3, Matlab prototype implementations for the two algorithms are developed, and tested against a Matlab built-in KNN classifier on the M&Ms dataset.

These algorithms will be ported to C++ for the implementation on the Zynq-7000.

The prototype algorithms perform as the Matlab benchmark, as shown Table 3. The two prototypes are considered OK, and the same performance will be required when running on the SoC.

Table 3: Comparison of prototype algorithms versus Matlab built-in benchmark implementation

Model	Own implementation	Matlab built-in benchmark implementation
KNN $K = 1$	Accuracy = 97% Missed = {22, 48, 49}	Accuracy 97% Missed = {22, 48, 49}
KNN general, $K = 3$	Accuracy = 98% Missed = {49, 64}	Accuracy = 98% Missed = {49, 64}

The above data shows that setting the hyperparameter  $K = 3$  gives a slightly more robust classifier, and also underlines why the general algorithm is the main focus.

### 6.2.2 Computational complexity and memory operations

Consider the problem of computing the KNN with  $R$  reference vectors with feature-dimensionality of  $N$ , and finding  $K$  nearest neighbours. Using FF/BRAM and DMA for the general KNN (see Figure 11 in the design chapter), and perhaps just a register and AXI-Lite for the  $K = 1$  version, the two algorithms are compared on typical and asymptotic computational complexity and memory operations in Table 4.

Table 4: Comparison of computational complexity and memory operations for the two implemented algorithms

Algorithm	Computational operations	PL memory operations <sup>1</sup>
<b>KNN for <math>K = 1</math></b> <ol style="list-style-type: none"> <li>1. Compute sq.dist.</li> <li>2. Compare to previous best</li> <li>3. Save index and value for current best neighbour</li> <li>4. Return result from HW</li> </ol>	<ol style="list-style-type: none"> <li>1. <math>R</math> references times (<math>N</math> subtractions + <math>N</math> multiplications).</li> <li>2. <math>R</math> comparisons.</li> </ol> <p>All cases: <b><math>2NR+R \sim O(NR)</math></b></p>	<ol style="list-style-type: none"> <li>3. Much fewer than <math>R</math> cases where current dist is better than previous best: <math>&lt; R</math> register write operations.</li> <li>4. <math>K = 1</math> write operation to PS over AXI.</li> </ol> <p>Worst case: <b><math>R+1</math> cheap writes to a register</b></p>

<b>KNN for general <math>K</math></b> <ol style="list-style-type: none"> <li>1. Compute sq. dist</li> <li>2. Stores each computed sq.dist.</li> <li>3. Return result from HW</li> <li>4. Sort list to extract <math>K</math> nearest distances</li> <li>5. Find indices of <math>K</math> nearest neighbours from least distances</li> <li>6. Voting algorithm</li> </ol>	<ol style="list-style-type: none"> <li>1. <math>2NR</math> (as above).</li> <li>4. <math>R \log_2(R)</math>, as C++ uses Quicksort as standard.</li> <li>5. Naïve binary search <math>K</math> times: <math>K \log_2(R)</math>.</li> <li>6. Tally across <math>K</math> neighbours and find highest vote: <math>K</math> additions + <math>K</math> comparisons.</li> </ol> <p style="text-align: center;">Typical case:  <math>2NR + R \log_2(R) + K \log_2(R) + 2K</math>  <math>\sim O\{N(R + \log_2(R))\}^2</math></p>	<ol style="list-style-type: none"> <li>2. <math>R</math> expensive write operations to FF/BRAM.</li> <li>3. <math>R</math> expensive write operations to PS over AXI or DMA.</li> </ol> <p style="text-align: center;">All cases:  <math>2R</math> <u>expensive</u> write operations to FF/BRAM and DMA/AXI</p>
---	--	---

**Notes:**

1) Not considering the fetching of 1 query vector ( $N$  memory reads), and the  $R$  reference vectors ( $NR$  memory reads), which is similar for both algorithms.

2) As typically  $K \ll N \ll R$ , the contribution from  $2K$  and  $K \log_2(R)$  are disregarded asymptotically.

For the dataset used in this project, the general KNN with say  $K = 3$ ,  $N = 63$ , and  $R = 1024$ , will require about 139,000 calculations and 2048 expensive write operations (FF/BRAM and DMA/AXI), where the simpler KNN with  $K = 1$  will require about 130,000 calculations but only worst-case 1024 cheap writes (to a register).

Two conclusions:

- The general KNN has additional 9000 operations (+7%) for sort/search, which are handled by PS.
- The main difference is the much larger amount of expensive memory operations required for the general KNN.

Clearly it must be expected that the general KNN is slower in hardware than the  $K = 1$  version.

### 6.3 Analysis of hardware platform and constraints for design

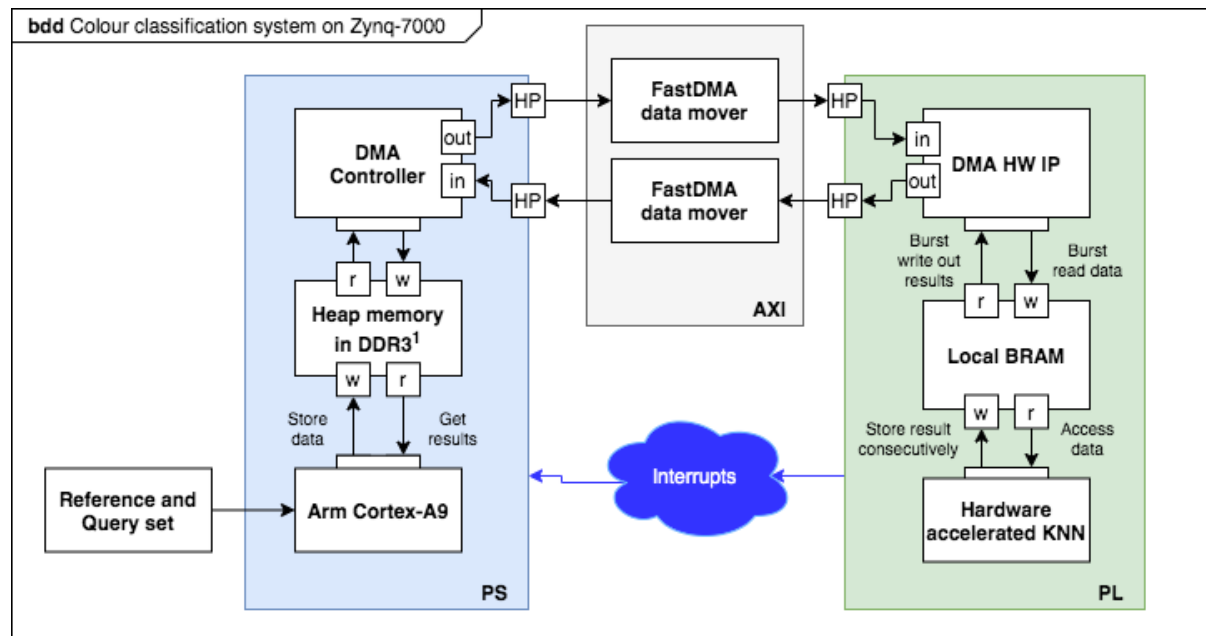
- PS: Zynq SoC processing system
  - Single-core Arm Cortex-A9 with SIMD instructions
    - Can handle complex computation, and OK to use for sorting/searching.
  - 512 MB DDR, 128 MB QSPI, 8 GB eMMC
    - Plenty for storing the dataset however we choose to implement it.
- PL: Artix-7
  - LUTs: 14400
    - Footprint of logic must be kept relatively small.
    - Fabric size too small for large DPU, too small for large CNN
  - DSP slices: 66
    - These are most likely ideal for the MAC operations in the KNN.
  - BRAM: 100 x 18K

## 7 Design

The key design goal for the system is to obtain faster performance than software-alone by accelerating the KNN algorithm in hardware, and therefore the design must:

- Optimize movement of data in/out of PL.
- Optimize concurrency in the KNN hardware.
- Subject to design constraints from the available resources in the z7007s, particularly LUTs and BRAMs available in the FPGA.

Figure 11 shows the system design as a block diagram on a conceptual level, based on the team's assumption that DMA is the fastest way to move data in and out of the PL, and assuming that BRAM is the best way to store and handle data locally in the PL. The hardware accelerated KNN algorithm is seen in the PL. The design space exploration later in this chapter is a process to determine the optimal design for a fast accelerator.



**Note:**

1) The DDR3 memory is physically located outside the PS, but the DDR3 controller is inside the PS, so DDR3 memory is modelled as inside the PS.

Figure 11: The baseline block diagram of the system using PS for processing tasks and PL hardware acceleration

To generate a system like this, SDSoc will be used to generate the acceleration block, the interfaces and the data movers, starting from a base platform created in Vivado.

### 7.1 Base hardware platform in Vivado and SDSoc platform

To build a base hardware platform, the guide from Avnet in ref. [11] and the related lab exercises in [12] are particularly helpful for MiniZed<sup>4</sup>, and a more general end-to-end

<sup>4</sup> They show how to build the hardware platform and to implement a hardware accelerated 24x24 matrix multiplication on MiniZed. The platform is built in Lab 2.

description is available from Xilinx in [2]. In short, a base hardware platform for SDSoc development requires:

- The PS7 processing system (must be configured).
- A concatenation (xlconcat) used to connect interrupt lines from the PL to the PS.
- Proc-system-reset blocks to handle reset conditions for blocks that will be added later.

The block design containing the stated elements, created with Vivado, is shown in Figure 12.

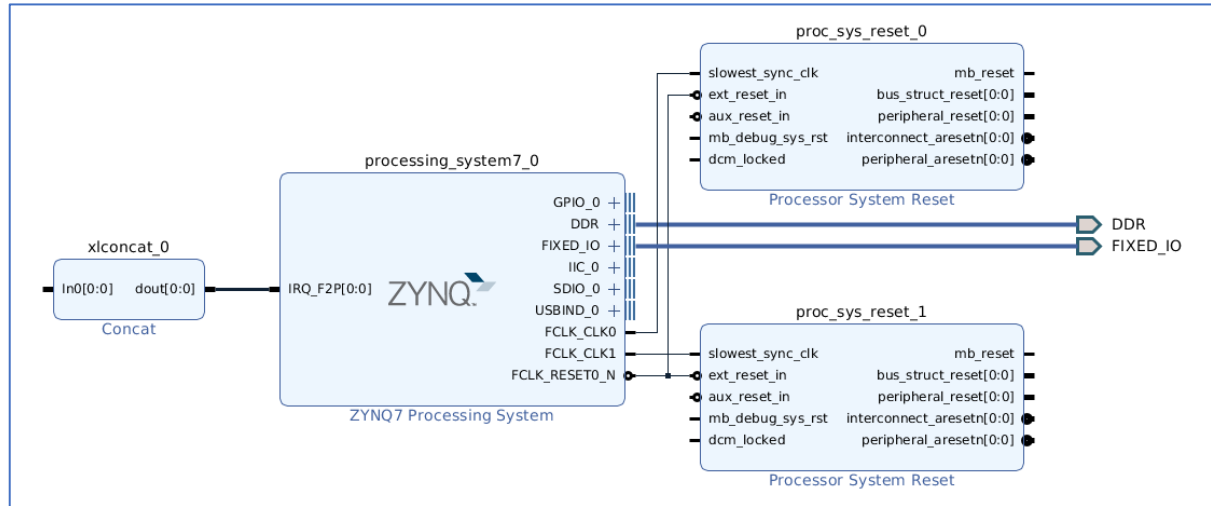


Figure 12: Hardware platform block design using Vivado IP Integrator

The processing system (PS7) is configured with activated peripherals as in Figure 13. The PL clocks are set at 50 MHz and 100 MHz for CLK0 and CLK1 respectively<sup>5</sup>.

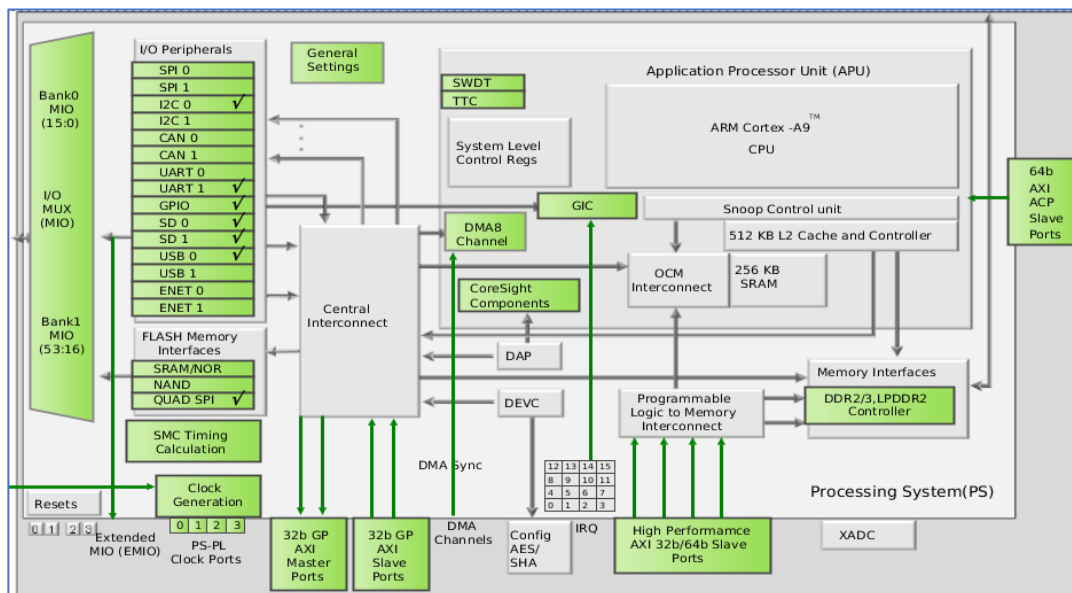


Figure 13: Peripheral configuration of PS7

<sup>5</sup> As recommended in [12]. The PS clocks the PL, as the PL has no other CLK input, and the clock rates into the FPGA is set using PLL multipliers to 50 MHz and 100 MHz. This is a design choice that can be tweak to optimize the system further later by increasing the clock rate. The PL fabric should be capable of up to 250 MHz, while some cores, e.g. the AXI-DMA IP, are supposedly only capable of up 200 MHz operation.

A list of TCL commands must be run to declare available interfaces, clocks, etc. and create a full hardware specification (DSA). The TCL commands are given in ref. [12, pp. 5-6 (lab 3)] and are explained in ref. [2, pp. 37-67].

Using the DSA from Vivado and the SDSoC Platform Generator, an SDSoC platform is created for this project. It is designed to run bare-metal (standalone) and will create an FSBL for deployment and BSP for the application (it is modified every time the accelerator design is changed)<sup>6</sup>. Conceptually, the SDSoC platform is illustrated in Figure 14.

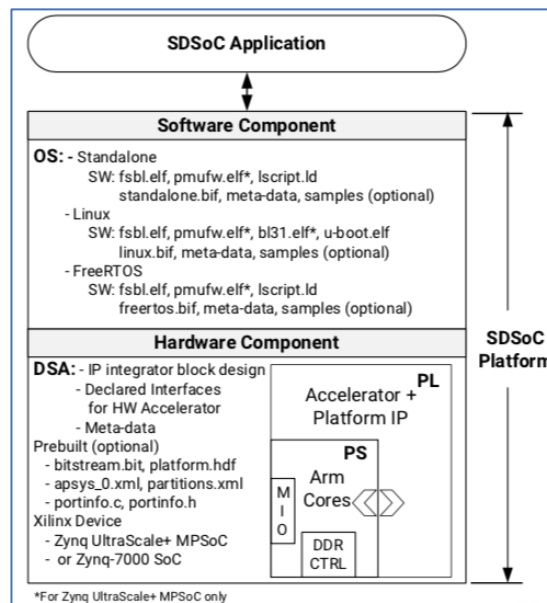


Figure 14: Components in the SDSoC platform [2, p. 5]

## 7.2 C++ algorithms and partitioning for PS and PL

The **acceleration target**, and also the software comparison, is the C++ implementation in Figure 15, based on the prototype developed and tested in Matlab in the “Analysis” chapter.

```

67 void knn_sw(const uint8_t* ref_vecs, // "size" reference vectors each of length "dim"
68             const uint8_t* query,   // 1 query vector of length "dim"
69             unsigned int* d2,        // allocated space to store sq. Euclid. dists.
70             int size,                // number of reference vectors
71             int dim)                 // number of features in a vector
72 {
73
74
75     unsigned long sum_dist;
76
77     // Nested loop to go through all reference vectors
78     for (int i = 0; i < size; i++) {
79         sum_dist = 0;
80
81         // MAC operation to go through all vector components
82         for (int j = 0; j < dim; j++) {
83             sum_dist += SQ(query[j] - ref_vecs[i*dim + j]);
84         }
85
86         // Store the squared Euclidian distance
87         d2[i] = sum_dist;
88     }
89 }

```

Figure 15: C++ computation of Euclidian distance between one query vector and many reference vectors.

<sup>6</sup> A similar reference design can also be downloaded from [19].

The full implementation code is in appendix 15.4.

The implementation also contains:

- The voting algo. which finds the  $K$  nearest neighbours based on the vector of squared Euclidian distances. This is a sort/search problem
- The scoring / accuracy algorithms, which test the KNN predictions against data labels and provide misclassifications and accuracy scores. This is a low-complexity problem.
- Simple user interface.

These three functions are **not targeted for acceleration** as they are not (relatively) computationally demanding, and are therefore partitioned onto the PS. All the algorithms are necessary to use the KNN, and to prove it correct. They have also been developed during the project, and the code is in appendix 15.4.4.

### 7.3 Profiling and performance measurement

It is already decided that the KNN algorithm squared distance computation is the part of the application code that is suitable for hardware acceleration. So the primary profiling / performance measurement used in this project is a simple method described in [13, p. 12]: Counting duration in clock cycles to execute different parts of the program/system. SDS has an API for doing this.

Comparison of absolute clock-cycle counts gives indication of changes in system performance from design changes and optimization attempts. The relative ratio between clock cycle counts during HW and SW execution of the *same function* gives an estimate of the achieved hardware speed-up<sup>7</sup>.

For finding bottlenecks in the hardware, *“the loop initiation interval (II) is another important performance metric. It is defined as the number of clock cycles until the next iteration of the loop can start.”* [14, p. 45]. Warning on II are found in the guidance tab during HLS. Labels are added to loops, so guidance from the SDS and HLS will be annotated with these.

Differences in power usage, fabric area or utilization of fixed resources in the FPGA are not considered central criteria in this project, even though relevant criteria in other projects.

### 7.4 Accelerator design

After the DSE described in a later section, the final system design spec is as follows:

- 100 MHz clocking of accelerator, and 100 MHz data motion network.
- DMA memory interface between PS and PL.
  - Sequential access pattern.
  - Vector components stored contiguously and transferred using uint8\_t.

---

<sup>7</sup> It is just an estimate, as the program for the PS is compiled using -O0, and as mentioned earlier, the FPGA is not running at the maximum clock-rate.

- Local FPGA storage of the query vector in FFs: 8 FFs per uint8\_t.
- Pipelining all loops.
- Unrolling loops with a factor of 32 (50%).
- Cyclic array partitioning with a factor of 32.
- MAC structure for algorithm, leveraging DSP48.

This design runs a single classification problem in about 443,000 CPU cycles using random numbers and 64 features (as DSE was done with this setting). This is 7.46x faster than the equivalent software algorithm. The DSE has increased the accelerator performance by 5.3x versus the starting point<sup>8</sup>. As mentioned earlier, there is potential to increase the FPGA clock rate to 200 MHz, which would yield almost 15x speed-up over software.

The *final* final system uses 63 features instead of 64, to be in line with the dataset. The resource utilization in the FPGA as computed during synthesis is in Figure 16. Notice the estimate of 17 DSPs and zero BRAMs. The system uses FFs instead as these are faster.

Utilization Estimates					
Summary					
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	17	-	-	-
Expression	-	0	0	2619	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	301	-
Register	-	-	1635	-	-
Total	0	17	1635	2920	0
Available	100	66	28800	14400	0
Utilization (%)	0	25	5	20	0

Figure 16: SDSoC estimate of resource utilization for final accelerator design.

The data motion, ports and data transfer characteristics are in Figure 17. The transfer cost is 1.7 CPU cycles / byte<sup>9</sup>. Also fewer bytes are transferred than in the baseline system.

Data Motion Network						
Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
knn_hw_1	ref_vecs	ref_vecs	IN	(size)*1	• length:(1024*63)	processing_system7_0_S_AXI_ACP:FastDMA
	query	query	IN	(size)*1	• length:(63)	processing_system7_0_S_AXI_ACP:FastDMA
	d2	d2	OUT	(size)*4	• length:(1024)	processing_system7_0_S_AXI_ACP:FastDMA
Accelerator Callsites						
Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Datamover Setup Time(CPU cycles)	Transfer Time(CPU cycles)
knn_hw_1	main.cpp:70:3	ref_vecs	(1024*63) * 1	contiguous	999	108678
		query	(63) * 1	contiguous	1038	1803
		d2	(1024) * 4	contiguous	1012	7990

Figure 17: Data motion overview from SDSoC

<sup>8</sup> The baseline accelerator runs the problem in 2.35 million CPU cycles which is a speed-up of 1.57x over the software. This design is based on: (1) The software algorithm in Figure 15, marked to run accelerated in SDSoC, (2) 50 MHz clocking of accelerator, and 50 MHz data motion network, (3) Random access memory mapped DDR interface from PL to PS (PL is master and commands read and write operations, (4) Transferring data in an array of 1024 reference vectors with 64 features each, each feature stored as an int.

<sup>9</sup> Was 3.3 CPU cycles / byte in the baseline.

The final system interface is specified in the header file shown in Figure 18. Notice the compiler directives instructing the system compiler to infer the specific interfaces.

```

1  #ifndef KNN_H
2  #define KNN_H
3
4  #include <stdint.h>
5
6  #define DATA_DIM 63    // features in a feature vector
7  #define DATA_SIZE 1024 // size of the reference (training) set
8
9
10 // copy will infer DMA (FastDMA) transfer to PL (from contiguous mem area!), and specify copy sizes
11 // data_access is strictly required to be sequential for speed-up
12 #pragma SDS data copy(ref_vecs[0:DATA_SIZE*DATA_DIM], query[0:DATA_DIM], d2[0:DATA_SIZE])
13 #pragma SDS data access_pattern(ref_vecs:SEQUENTIAL, query:SEQUENTIAL, d2:SEQUENTIAL)
14 #pragma SDS data mem_attribute(ref_vecs:PHYSICAL_CONTIGUOUS, query:PHYSICAL_CONTIGUOUS, d2:PHYSICAL_CONTIGUOUS)
15 void knn_hw(uint8_t* ref_vecs, uint8_t* query, unsigned int* d2);
16
17 #endif

```

Figure 18: Baseline system interface design, with memory mapped (zero\_copy) random access to DDR from PL. File knn.h.

The final accelerator algorithm in Figure 19 is *functionally* similar to the software KNN, but structurally different, notice in particular the compiler directives to direct HLS to infer specific concurrency and scheduling.

```

6 void knn_hw(
7     uint8_t *ref_vecs, // Reference vectors (training data)
8     uint8_t *query,    // Query vector (test vector)
9     unsigned int *d2    // Sq. Euclidian distances return value
10 )
11 {
12
13     // BRAM block for local memory
14     uint8_t query_local[DATA_DIM];
15
16     // Partition the local BRAM for more concurrency. A high factor puts data in FFs.
17     #pragma HLS ARRAY_PARTITION variable=query_local cyclic factor=32 dim=1
18
19     // Running sum
20     unsigned int sum_dist = 0;
21
22     // Get entire query vector (from DMA) to local BRAM to be read multiple times
23     get_query: for(int i = 0; i < DATA_DIM; i++){
24         #pragma HLS PIPELINE
25         query_local[i] = query[i];
26     }
27
28     // Perfect nested loop
29     for_each_ref_vec: for(int i = 0; i < DATA_SIZE; i++) {
30         compute_dist: for(int j = 0; j < DATA_DIM; j++) {
31             #pragma HLS PIPELINE
32             #pragma HLS UNROLL factor=32
33
34             // MAC operation to compute sq. Euclidian distance, getting ref_vecs sequentially
35             sum_dist += SQ(query_local[j] - ref_vecs[i*DATA_DIM + j]);
36
37             if(j == DATA_DIM-1) {
38                 d2[i] = sum_dist; // send final sq. dist. to i-th ref.vec.
39                 sum_dist = 0;     // reset
40             }
41         }
42     }
43 }

```

Figure 19: Final accelerator algorithm, functionally similar to the software implementation. File: knn.cpp.

The memory is allocated contiguously as shown in Figure 20, based on [15, p. 54].

```

42 // Heap allocation
43 uint8_t* ref_vecs = (uint8_t*) sds_alloc(sizeof(uint8_t) * DATA_SIZE * DATA_DIM); // Reference (train) set
44 uint8_t* query    = (uint8_t*) sds_alloc(sizeof(uint8_t) * QUERY_SIZE * DATA_DIM); // Query (test) set
45 unsigned int* d2_hw = (unsigned int*) sds_alloc(sizeof(unsigned int)*DATA_SIZE); // 1024 return vals
46
47 // Heap allocation for PS algo.
48 unsigned int* d2_sw = (unsigned int*) malloc(sizeof(unsigned int)*DATA_SIZE);

```

Figure 20: Memory allocation using the SDS API ensures contiguous memory regions are allocated in DDR. File: main.cpp.



## 7.5 Design space

The development of the system outlined above was guided by technical manuals and some knowledge of the peripherals on the Zynq platform. There are many opportunities for optimization, e.g. to speed up data transfers, and to get better concurrency by using more DSP blocks in parallel. Memory and loops can be optimized to make this happen, and a few of the concepts are briefly mentioned in the following sections.

### 7.5.1 Loop optimization

Loops are by default synthesized as sequential. So, several tools are available to get better concurrency in hardware, see e.g. the HLS optimization guide [16] or the SDSoC optimization guide [13]. The first is pipelining, which is requested using the `#pragma HLS PIPELINE` directive. The effect of pipelining is shown in Figure 21.

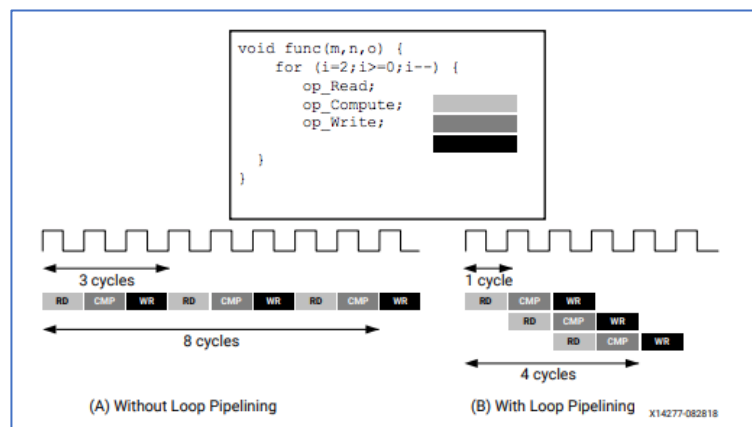


Figure 21: Comparison between sequential and pipelined loop [16, p. 116].

The second is loop unrolling, which can be done completely, or partially by `#pragma HLS UNROLL factor=<N>` [16, p. 125]. This is illustrated in Figure 22.

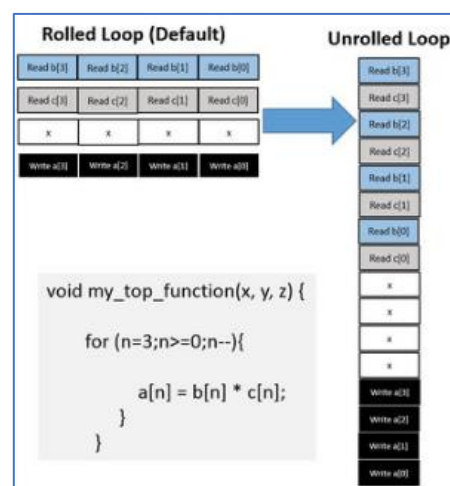


Figure 22: Unrolled loops (if there are no loop-carried dependencies) will give completely concurrent operations.

### 7.5.2 Local memory and array partitioning

BRAM has two ports, and this can be a bottleneck preventing multiple concurrent reads/writes. So, arrays (representing memory) can be partitioned. Cyclic partitioning makes most sense here, and is illustrated in Figure 23.

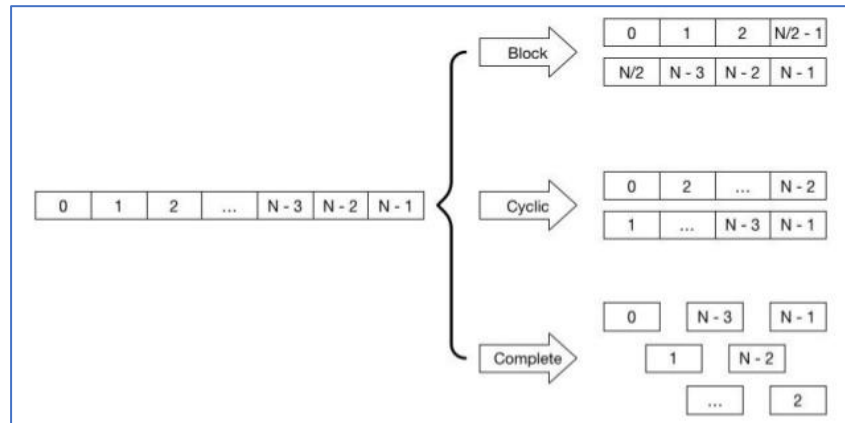


Figure 23: Array partitioning splits memory into different blocks, allowing more concurrent reads/writes.

### 7.5.3 Algorithm design: Scheduling and binding

HLS is responsible for scheduling and binding, given the code and directives supplied. So for a MAC operation like implemented in the KNN, Figure 24 is an illustration of how HLS will schedule and bind.

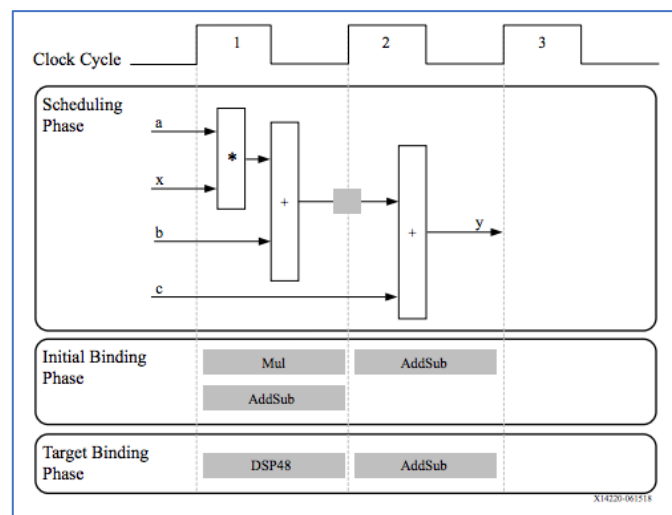


Figure 24: HLS schedules operations and binds to resources like DSP blocks, based on code and directives given [17].

### 7.5.4 Problem size

Using the simpler  $K = 1$  algorithm earlier in the DSE, different problem sizes were considered to get an overview of the attainable speed-up. Table 5 shows the difference.

Table 5: Comparison of memory interface and problem size

Memory interface and data motion	Problem size ( $R \times N$ )			
	1024 vectors x 3 features	1024 vectors x 16 features	1024 vectors x 32 features	1024 vectors x 64 features
Direct interface PS-PL. <ul style="list-style-type: none"> <li>• Accesses reference vector data in shared DDR (random access).</li> <li>• Burst-reads the query vector one-time to local BRAM.</li> </ul>	1.67	2.53	3.83	5.26
Project: <i>knn_classifier_vector_accelxx</i>				

**Note:** The numbers are obtained using a 100 MHz clock frequency in the PL, and a 100 MHz data motion network. With a 50 MHz data motion network, the speed-ups are 0.5 times the stated values, e.g. 3 features with direct interface is at 0.84 and is slower than software. Speed-up is based on the average over 10 runs each of the software and hardware implementations, and computed as  $speedup = avg\_cpu\_cycles\_sw / avg\_cpu\_cycles\_hw$ .

Table 5 shows that the gain in speedup increases with problem size. The gain is not a linear function, in part due to the overhead of setting up the memory access first-time, and as other bottlenecks arise, such as concurrent memory access and dependencies.

## 7.6 Overview of the design space exploration

Pragmas (compiler directives marked by `#pragma`) allow control of how the system and hardware is synthesized from high-level code. A full reference of pragmas is available in [18]. Specifically, SDS and HLS pragmas are useful for this project:

- SDS pragmas are for the system compiler (system interconnections, timings, buffers, data movers, etc.).
- HLS pragmas are for the high-level synthesis compiler (loop unrolling and optimization, array partitioning, etc.).

The table below outlines the design parameters that were explored in the DSE. The process was a systematic search for a more performant design among a very large number of combinatorial possibilities. There are still many unexplored optimizations.

Table 6: The design space exploration process is summarized by the list of design parameters that were investigated during the search for a more performant system.

Design parameter	Optimization trials and experiments to reach the final design
<b>Clock rate</b>	<b>Accelerator and data motion clock rate</b> <i>Step:</i> Set clock rate to 100 MHz up from 50 MHz. <i>Effect:</i> If there are no other bottlenecks -> twice the speed-up. <i>Conclusion:</i> Linear speed-up effect from FPGA clocking.
<b>Data motion network</b>	<b>Streaming memory mapped interface</b> <i>Step:</i> Set the interface to <code>zero_copy</code> with sequential access pattern. Requires

	<p><i>Effect:</i> If the access is strictly sequential and there are no other bottlenecks, an AXI-streaming interface is inferred, which is significantly faster than random access,</p> <p><i>Conclusion:</i> Useful, but DMA is faster.</p>
	<p><b>DMA transfer between PS and PL</b></p> <p><i>Step:</i> Set the interface to copy with sequential access pattern [18, p. 50].</p> <p><i>Effect:</i> The copy pragma requires that memory is copied between DDR and accelerator. Inferred data movers are FastDMA IP using AXI_ACP-ports. Significant improvement in terms of transfer cost.</p> <p><i>Conclusion:</i> Best memory interface for the problem size, according to [13, p. 24], transfers of more than 300 bytes is best handled by DMA.</p>
	<p><b>Local memory between DMA and HW logic</b></p> <p><i>Step:</i> Insert memory element for repeated access to query vector array without violating DMA strict sequential access. Design in Figure 19, lines 23-26.</p> <p><i>Effect:</i> Maintains high performance of DMA while allowing the algorithm to work.</p> <p><i>Conclusion:</i> Required.</p>
	<p><b>AXI FIFO for return array</b></p> <p><i>Step:</i> Require FIFO data mover to return results for small problems.</p> <p><i>Effect:</i> According to [13, p. 24], transfer of less than 300 bytes is well-handled by a FIFO. The KNN <math>K = 1</math> algorithm however is better when using a scaler over AXI-Light.</p> <p><i>Conclusion:</i> Not required for this design.</p>
<b>Data bitwidth optimization</b>	<p><b>Narrower unsigned datatype</b></p> <p><i>Step:</i> Change datatype for data arrays from int to uint8_t.</p> <p><i>Effect:</i> More vectors can be stored, fewer clock cycles to transfer over DMA. Arm Cortex-A9 processes the data faster also. When using ints and full unrolling, it occurred that the FPGA ran out of resources. Doesn't happen with narrower bitwidths.</p> <p><i>Conclusion:</i> Good optimization for this problem, used in final design.</p>
<b>Loop optimization</b>	<p><b>Pipelining loops</b></p> <p><i>Step:</i> Require pipelining using #pragma HLS pipeline. Be careful where they are placed, as they will make the design bloated and slow if placed at the outer loop Figure 19, lines 29. See [16, pp. 17-20] for details.</p> <p><i>Effect:</i> Higher concurrency in the design, faster accelerator.</p> <p><i>Conclusion:</i> Good optimization, used in final design.</p>
	<p><b>Perfect nested loop<sup>10</sup></b></p> <p><i>Step:</i> Assignments, branching and conditionals must be moved inside inner loops, as done in Figure 19, lines 37-39.</p> <p><i>Effect:</i> Perfect nested loops can be flattened by HLS (multiple indices handled simultaneously), which in turn gives other optimizations. Noticeable difference after moving the conditionals.</p> <p><i>Conclusion:</i> Good optimization, used in final design.</p>
	<p><b>Array Partitioning</b></p> <p><i>Step:</i> See [18, p. 58] and also see SDS data access patterns [18, p. 33]. Require cyclic partitioning.</p> <p><i>Effect:</i> There is a trade-off in the factor of partitioning. Too small does not have enough effect, and too much slows the design down due to large mux'es being inferred.</p> <p><i>Conclusion:</i> A factor of around 32 seems to be a sweet spot for this design. Used in final design.</p>
	<p><b>Partially unrolling loops</b></p> <p><i>Step:</i> Unrolling loops with factor 16 and 32. Don't unroll loops that read from DMA.</p> <p><i>Effect:</i> Factor 32 gives better performance than factor 16, and both better than no unrolling. Also see Array partitioning, as these factors are connected.</p> <p><i>Conclusion:</i> Good optimization, used in final design.</p>
	<p><b>Fully unrolling loops</b></p>

<sup>10</sup> A perfect nested loop can be flattened by HLS. Requirements for a perfect loop are: (1) Only the inner loop has a loop body, (2) There is no logic or operations specified between the loop declarations, and (3) all the loop bounds are constant [15, p. 38].

	<p><i>Step:</i> Unroll with factor 64 (same as complete)</p> <p><i>Effect:</i> Implements memory as flip-flops. Guidance however reports that large mux'es might be slow. Fully unrolling loop disables pipelining.</p> <p><i>Conclusion:</i> Full unrolling is not useful for this design, as the performance is worse than factor 32.</p>
<b>Algorithm design</b>	<p><b>Tapped-delay line instead of MAC structure</b></p> <p><i>Step:</i> Use a tapped delay line structure to avoid read and write of same variable in one step as the MAC operation does. Idea: <math>\text{buf}[j+1] = \text{buf}[j] + (q_j - r_j)(q_j - r_j)</math>, with <math>\text{buf}[0] = 0</math>.</p> <p><i>Effect:</i> Slightly slower, likely due to BRAM r/w operations requiring more cycles than MAC ops.</p> <p><i>Conclusion:</i> Does not add value in this design.</p>
	<p><b>Remove loop-carried dependence to parallelize tapped delay line</b></p> <p><i>Step:</i> Add the <code>#pragma HLS DEPENDENCE variable=buf array inter false</code>.</p> <p><i>Effect:</i> Result is wrong (as expected, it is wrong to remove the dependence, as <math>\text{buf}[j+1]</math> should not be computed before <math>\text{buf}[j]</math>).</p> <p><i>Conclusion:</i> Don't do this!!</p>
<b>DDR memory attributes</b>	<p><b>Non-cacheable contiguous memory allocation</b></p> <p><i>Step:</i> Allocate non-cacheable mem. using <code>sds_alloc_non_cacheable</code>, see API in [15, p. 54].</p> <p><i>Effect:</i> No significant speed-up in the PL operation was noticed, but there was a drastic slow-down when the PS needed to process the non-cacheable data.</p> <p><i>Conclusion:</i> Don't do this!! Use this only if the memory region in DDR is not used by the processor, e.g. exclusively used by the accelerator or between hardware peripherals.</p>

## 7.7 Design learning from the process

Key learning from the DSE process:

- It is extremely nice to have high-level abstractions to do rapid prototyping.
- Be systematic and expect to use several days to iterate through different designs.
- Read guidance from HLS during cross-compilation and synthesis, e.g. focus on initiation interval violations that point to where dependencies in the code can potentially be refactored.
- Optimize for overall system performance:
  - Learned the hard way: Initially looked only at relative speed-up, and e.g. one optimization to make heap memory non-cacheable to transfer faster to PL seemed to give 33x speed-up, but in fact was mainly due to a slowdown of PS.
- Use arrays so data is read and written sequentially between PS/PL
  - If not possible, put local memory (FF/BRAM) in between.
- Control the factor of loop unrolling, to get some unrolling without taking all resources.
- Place pipeline directives at the level below input units (here we input vectors, so we pipeline processing the vector components).
- Partition arrays to get multiple memory blocks: Faster system because BRAM only has two ports, so more loop unrolling can be done if memory access is split over several BRAM blocks. If doing high-factor partitioning, HLS will use FF for memory, which is faster than BRAM, but only has 1 port. All this is wasteful in terms of FPGA area.
- Branching and conditional statements inside loops is OK for FPGA implementation because nested loops can be flattened by HLS.
- Small changes in logic can have large changes in cross-compile and synthesis times, e.g. going from 12-14 minutes to 20 minutes by moving an assignment statement from outside a loop to inside a conditional statement in a loop.

## 8 Test, deployment and final implementation

### 8.1 Test and deployment

A screenshot from the implemented system can be seen in Figure 24. Notice that the KNN results and verification reports confirm that SW and HW accelerator give the same results, and that these are similar to the Matlab prototypes from the “Analysis”.

Notice also that the speed-up here is a factor 6.9x, below the 7.4x found during DSE. This is likely because changing the feature dimension from 64 to 63 gave a different synthesis with different performance characteristics.

Finally, notice from the screenshot that the functional partitioning of SW and HW allows a simple user interface, where the  $K$ -value for the classification can be chosen freely.

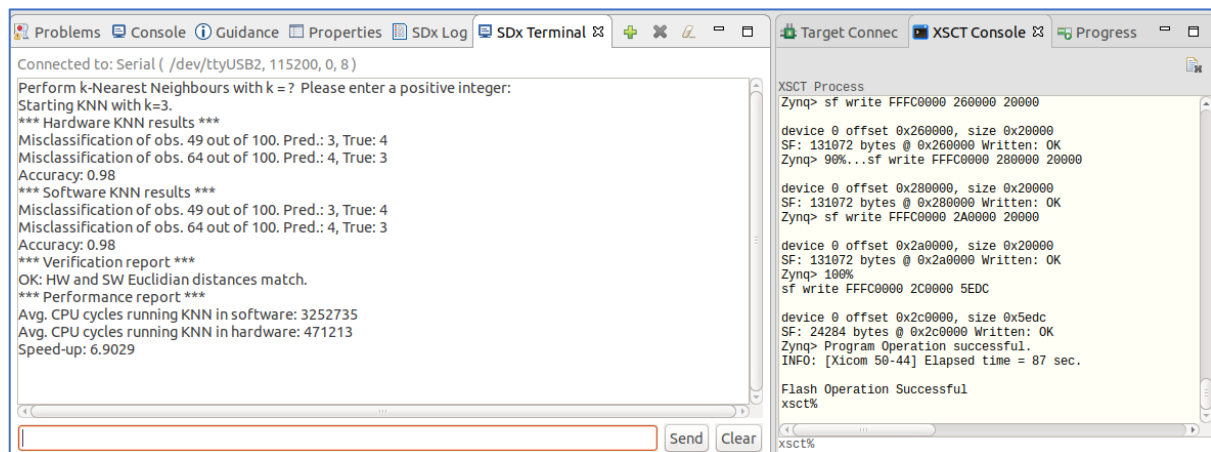


Figure 25: Test runs, verification and performance comparison for the implemented system

The right side of Figure 25 shows the Xilinx Software Command-line Tool (XSCT), which is used to deploy a permanent image (boot.bin) and FSBL to the device flash memory. The device is flashed using the two commands in Figure 26.

```
cd <project_dir>/Debug/sd_card/
exec program_flash -f boot.bin -fsbl <platform_location>/sw/sysconfig1/boot/fsbl.elf -
flash type qspi single
```

Figure 26: XSCT commands to deploy a standalone bootable image to Zynq-7000

The boot.bin file is 2.9 MB and the FSBL is 558 kB. As the FSBL is larger than the 256 kB device OCM, the device boots using XIP mode over QSPI, which underlines the importance of including the QSPI interface in the original PS7 config.

### 8.2 Final SoC implementation

The final block design in the IP Integrator is shown in Figure 27. The parts from the base hardware platform can be found, in particular the Zynq PS, the concatenation block for IRQ now fully connected, and one of processor system resets connected to peripherals (DMA) and the AXI bus system.

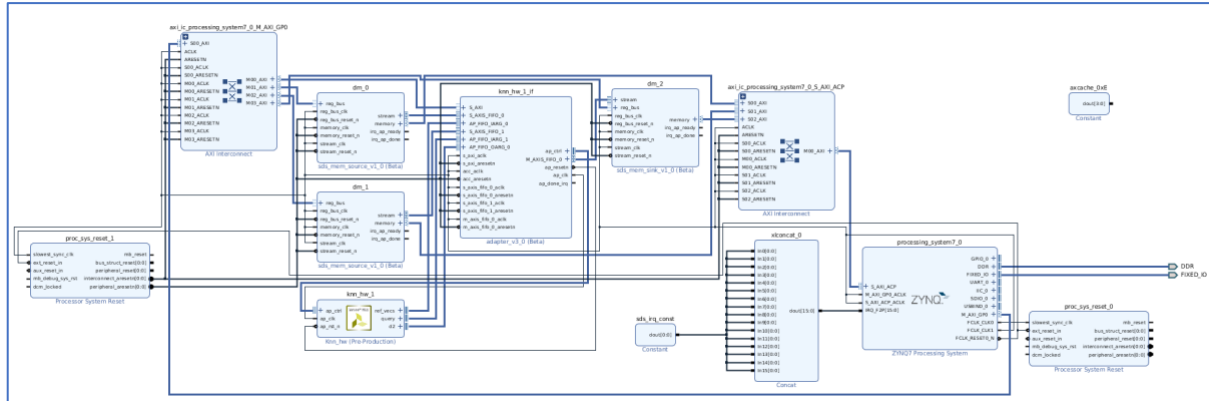


Figure 27: Vivado IP Integrator Block Design view of KNN accelerator system

A cutout of the figure is shown in Figure 28. Here the HLS generated block (knn\_hw\_1) and the data movers generated by SDS are visible. In particular, notice that input data is fed in/out to/of the HLS block by FIFOs (hence sequential access), and that the data movers (DMA) are connected to memory and stream data into the KNN's interface (knn\_hw\_1\_if).

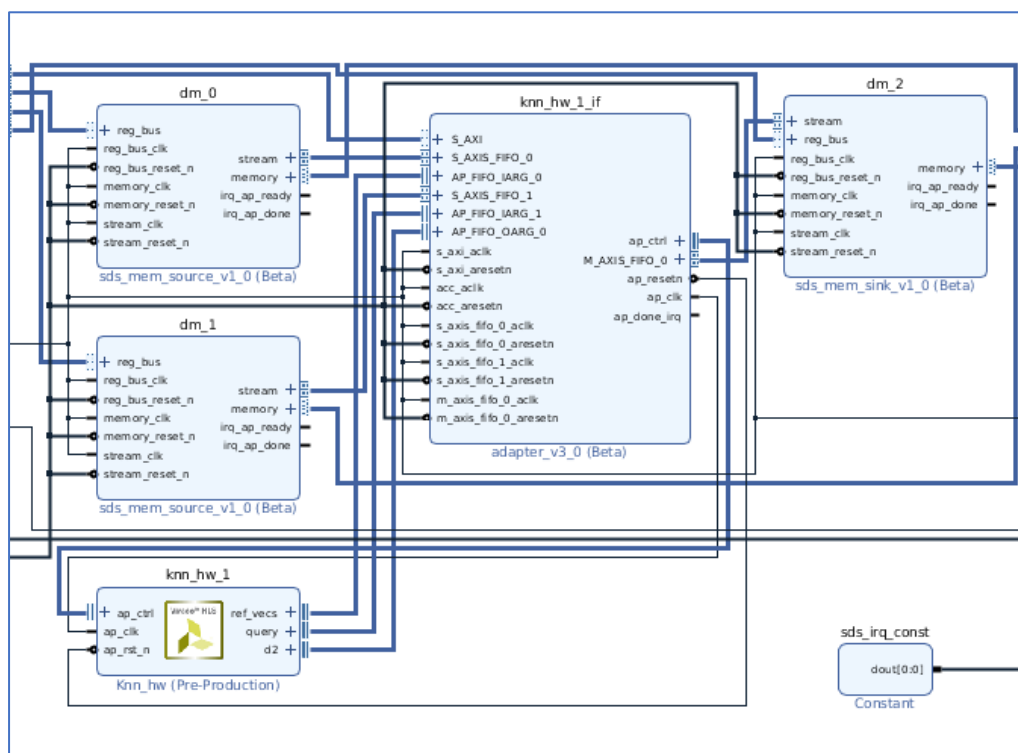


Figure 28: Selection of the block design showing the HLS block and connections

The implemented hardware fully makes sense given the directives given in SDSoc.

Figure 29 shows that the KNN hardware block actually uses 32 DSPs and no BRAMs, which is in line with the loop unrolling requested (and higher than the SDSoc estimate of 17 DSPs).

Name	^1	Slice LUTs (14400)	Slice Registers (28800)	F7 Muxes (8800)	Slice (4400)	LUT as Logic (14400)	LUT as Memory (6000)	Block RAM Tile (50)	DSPs (66)	Bonded IOB (54)	PHY_CONTROL (2)	BUFIO (8)
> knn_hw_1 (mz_avnet_		795	973	0	352	795	0	0	32	0	0	0

Figure 29: Actual utilization from Vivado

The floorplan in Figure 30 of the implemented design also confirms that there is space for more on the FPGA, e.g. another KNN block, as there are also still free DSPs.

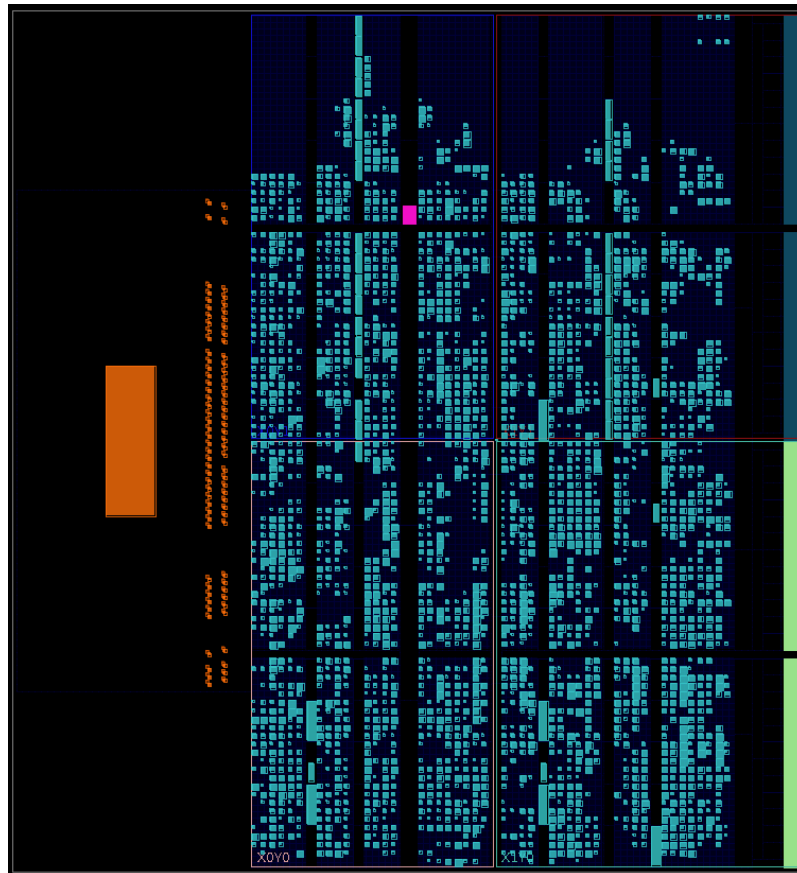


Figure 30: Floorplan of the implemented design from Vivado



## 9 Verification

This section verifies the implementation of the requirements.

Table 7: Requirements specification and verification tests for the system-to-be

Req.	Requirement description	Verification test
1	The system must be able to separate more than two colours from the M&Ms colour palette.	Run a test data set containing three colours on the system. Algorithm must have same accuracy as benchmark KNN algorithm in Matlab using same data.  <b>Result: OK! The system handles all 5 colours with the same accuracy and same misclassifications as the Matlab benchmark algorithm.</b>
1-A	<i>Nice-to-have improvement:</i> All 5 colours in the M&Ms colour palette can be classified.	Same with all 5 colours.  <b>Result: OK! See above. The dataset contains all 5 colours, and there are no colours that cannot be classified.</b>
2	The system must take as input sampled image patches of RGB colour. The input is structured as N-dimensional feature vectors. The system must take R reference vectors (training data) and Q query vectors (to be classified), with $R \gg Q$ .	Generate a dataset with 1024 reference vectors and 50-60 features describing RGB samples in an image patch. Test the dataset using a benchmark algorithm and require the same performance from the Zynq system.  <b>Result: OK! The system takes 1024 vectors with 21 RGB samples (63 values). The 21 triplets are sampled from an image patch, including some noise.</b>
3	Input is stored on the eMMC	No test.  <b>Result: OK! The data is stored in flat text files, which are loaded into the system by way of header files. These are compiled into the system, and are deployed to the flash memory using QSPI.</b>
4	The system must output the recognised colour as text or code to a terminal.	Reverse functional test. The system outputs the errors when classifying labelled data. The errors must be human-readable in a terminal.  <b>Result: OK! Demonstrated in test section.</b>

As can be seen from the results in the table, all 4 must-have requirements, and the 1 nice-to-have, are implemented.

## 10 Discussion

Overall, the goals of the project were reached: both in terms of requirements and learning. It was an interesting process, and one of the key insights is how rapidly an accelerator can be developed and implemented in hardware using co-design and HLS. However, the first implementation will typically not be optimal, and another key learning is how much experimentation and exploration is needed to get to a meaningful level of acceleration.

### 10.1 Development experience

The hardest part of the project was the incompatibility between tool versions, and generally to bridge the knowledge gap between *what* we wanted to do and *how* to actually do it with the tools. The project turned out to be more difficult than first imagined, and in particular sufficient time has to be budgeted for experimentation and optimization. It is a slow process, as system hardware synthesis takes about 15-20 minutes, and flashing the device takes another 90 secs. So, a few errors can quickly cost hours.

## 11 Conclusion

In this project, we have implemented part of an ML-based vision system on a Zynq-7000 SoC, achieving around 7x acceleration versus using the Arm Cortex-A9. The system is implemented as per requirements. The system is relevant both as educational/illustrative project and in general as it solves a real problem.

We have also reviewed methodology and Xilinx tools for co-design. Further, we have investigated and documented a number of ways in which a high-level design can be optimized and synthesized to hardware using the Xilinx tools, to be in line with our design requirements.

## 12 Improvements and future work

A number of ideas for future work have come up during the development:

- A larger device with more floorspace for larger models. Based on conversations with Søren Høyrup from Avnet Silica (Xilinx distributor), the Ultra96 board would be an ideal candidate for ML applications.
- It would be obvious to next attempt to get a feed-forward network or even a CNN onto the PL.
- Algorithmic improvement: Currently, the accelerator is “restarted” for every call. It is inefficient to reload data using the DMA at every call, and some persistence should be built in.
- Integration with more parts of a vision system: First and most valuable would be to implement the image acquisition and pre-processing steps. Afterwards, some decision logic and actuation could be implemented, or some networked communication that streams results of the classification.

## 13 Terminology

Term	Definition
ACP	Accelerator Coherency Port. Low-latency interface to PS, where the PL is master.
AXI	Advanced eXtensible Interface. Bus system to connect PS with peripherals, or PS and PL.
BRAM	Block RAM. Memory component for storing larger amounts of data inside FPGA programmable logic.
BSP	Board Support Package. Low-level hardware-specific software to use core platform features.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
DDR3 RAM	Double Data Rate 3 RAM. High bandwidth memory following standard/spec. 3.
DPU	Deep-Learning Processor Unit. Xilinx IP block in the Vitis flow for implementing CNNs.
DSA	Device Support Archive. Component in an SDSoC platform that contains the block design from Vivado IP Integrator and the declared interfaces for the HW accelerator.
DSE	Design Space Exploration. Systematic exploration process of components, operation modes, interconnections, etc. that are possible in a system, in order to get closer to an optimal design.
DSLR	Digital Single-Lens Reflex. Digital camera.
DSP slice, DSP48	Digital Signal Processing slice/block. Hardware embedded in the FPGA fabric to accelerate MAC operations.
ELF	Executable Linkable Format. File format that can be loaded directly to memory, yielding instructions with correct memory addresses/offsets. Subsequently, instructions can be directly executed by a processor.
eMMC	Embedded Multi-Media Controller. Memory like an SD-Card soldered directly onto the board.
FF	Flip Flop. Latch, bistable memory element. Used for storing smaller amount of data in FPGA.
FSBL	First-Stage Boot Loader.
FPGA	Field-Programmable Gate Array.
HDL	Hardware Description Language. Such as VHDL, Verilog.
HLS	High-level synthesis.

	Design method for synthesizing high-level languages, e.g. C/C++, to RTL-level hardware descriptions.
HP	High Performance Port. Port type between PS and PL.
HW	Hardware.
IDE	Integrated Development Environment.
II	Initiation Interval. Number of clock cycles before the function can accept new inputs.
IP	Intellectual Property. In context of this report, it refers to hardware components that can be instantiated and re-used as blocks in a hardware design.
KNN or kNN	K-Nearest Neighbours. Machine Learning / supervised learning classification algorithm.
LUT	Look-Up Table. Hardware logic component for specifying output given certain set of inputs. Fundamental building block in FPGA logic.
MAC	Multiply and Accumulate. Mathematical operation used in digital signal processing.
ML	Machine Learning.
MM	Memory Mapped. Direct memory interface where a resource or peripheral is accessed directly via a memory address.
OCM	On-chip memory. 256 kB memory area in the PS7.
OS	Operating System.
PL	Programmable Logic. The FPGA fabric with connection ports to PS.
PS	Processing System. System component containing CPU core(s), cache, snoop control unit, DDR RAM controller, interrupt controller, QSPI memory, connectivity peripherals, and connection ports to PL.
QSPI Flash	Flash memory connected by a Quad Serial Peripheral Interface.
RAM	Random Access Memory.
ROM	Read-Only Memory.
RTL	Register Transfer Logic.
RTOS	Real-Time Operating System.
SDK	Software Development Kit. In this context, a Xilinx IDE for software development targeted at Xilinx hardware platforms. Offered as the main tool until mid-2019, now succeeded by Vitis and maintained as a legacy tool.
SDS	Software-Defined System. Either used with the pragmas for synthesis or to refer to the system compilers sdscc and sds++.
SDSoC	Software-Defined System On Chip. Xilinx's Eclipse-based IDE for HW/SW co-design for SoC platforms. Succeeded by Vitis in late-2019.

SoC / AP SoC	System on a Chip / All Programmable SoC.
SW	Software.
TCL	Tool Command Language. Scripting language for automation, e.g. in Vivado.
VHDL	Very High Speed Integrated Circuit Hardware Description Language. Hardware description language to describe integrated circuits, programmable logic such as FPGAs, etc.
Vitis	Xilinx's integrated tool for end-to-end SoC development since late 2019, includes Vivado, and succeeds SDK, SDSoC, SDAccel.
Vivado	Xilinx IDE for designing hardware platforms.
XIP	Execute-in-Place. Booting the system directly from the flash using the QSPI interface.

## 14 References

- [1] J. Teich, "Hardware/Software Codesign: The Past, the Present, and Predicting the Future," *Proceedings of the IEEE*, vol. 100, no. May 13th, pp. 1411-1430, 2012.
- [2] Xilinx, "UG1146: SDSoC Environment Platform Development Guide," ver. 2018.3, 2019.
- [3] Avnet, "HDL examples," [Online]. Available: <https://github.com/Avnet/hdl>.
- [4] Avnet, "SD-Card vhd code," [Online]. Available: [https://github.com/Avnet/hdl/blob/master/IP/minized/hdl/vhdl/sdcard\\_mgr.vhd](https://github.com/Avnet/hdl/blob/master/IP/minized/hdl/vhdl/sdcard_mgr.vhd).
- [5] xilinx, "Github: Runtime (XRT)," [Online]. Available: <https://github.com/Xilinx/XRT>.
- [6] Xilinx, "Vitis Unified Software Developmet Platform," 2020.1, 2020.
- [7] xilinx, "Github - Vitis AI ModelZoo," [Online]. Available: <https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo>.
- [8] Xilinx, "UG1414 - Vitis Ai User Guide," v.1.3 , 2020.
- [9] Xilinx, "UG1431 - Vitis AI User Guide," v1.2, 2020.
- [10] Xilinx, "PG338 - Zynq DPU Product guide," v.3.2, 2020.
- [11] Avnet, "A Practical Guide to Getting Started with Xilinx SDSoC (Speedway)," 07 Sep 2017.
- [12] Avnet, "A Practical Guide to Getting Started with Xilinx SDSoC, Lab 0-4," Ver. 2, tool ver. 2017.4, 2018.
- [13] Xilinx, "UG1235: SDSoC Profiling and Optimization Guide," Ver. 2018.3, 2018.
- [14] J. M. a. S. N. Ryan Kastner, "Parallel Programming for FPGAs," 2018. [Online]. Available: <https://arxiv.org/pdf/1805.03648.pdf>. [Accessed 01 12 2020].
- [15] Xilinx, "UG1278: SDSoC Programmers Guide," Ver. 2018.3, 2019.
- [16] Xilinx, "UG1270: Vivado HLS Optimization Methodology Guide," Ver. 2018.1, 2018.
- [17] Xilinx, "UG902: Vivado HLS," Ver. 2018.3, 2018.
- [18] Xilinx, "UG1253: SDx Pragma Reference Guide," ver. 2018.3, 2019.
- [19] Element14 (Adam Eberly), "MiniZed Reference Designs: SDSoC\_Platform\_v2019.1," 31 Oct 2019. [Online]. Available: <https://www.element14.com/community/docs/DOC-95639#dc-reference-designs>. [Accessed Nov 2020].

## 15 Appendices

### 15.1 Appendix: Matlab code for dataset generation

```

%% Generate a dataset for the Zynq-7000 SoC KNN
% Janus Bo Andersen, Nov/Dec 2020.

clear; clc; close all;

%%
% Load the M&Ms dataset instead of rebuilding it

%load('mm_dataset.mat');

%% Rebuild M&Ms dataset manually
% make dataStore of images
imds = imageDatastore('img/');
imgs = readall(imds);

%%
% Manually annotated colours for the 32 images in same order as images
label_names = {'blue','yellow','brown','red','blue','brown','yellow', ...
               'brown','red','yellow','blue','brown','yellow','red','green',...
               'red','blue','green','red','yellow','red','brown','red',...
               'red','brown','blue','brown','brown','green','brown','brown',...
               'yellow'};

%%
% Hashmap for the M&Ms to translate from name to id, e.g. 'blue' -> 1
% The 5 M&Ms colours and associated colour ids
key_set = {'blue','yellow','brown','red','green'};
id_set = [1,2,3,4,5];
M = containers.Map(key_set, id_set);

%%
% Transform the names to ids
label_ids = {};
for k = 1:size(label_names,2)
    label_ids{k} = M(label_names{k});
end

%%
% Manually crop the first 32 images and store label together

for k = 1:32
    [J, rect] = imcrop(imgs{k});    % manually crop kth image
    mm_dataset{k}.im = J;
    mm_dataset{k}.rect = rect;
    mm_dataset{k}.label_id = label_ids{k};
    mm_dataset{k}.label_name = label_names{k};
end

```

```

%%
% Augment the dataset with labels on top of the images for visualization
imgs_for_tile = {};

for k = 1:size(mm_dataset,2)
    mm_dataset{k}.im_label = insertText(mm_dataset{k}.im, [10 60], ...
        mm_dataset{k}.label_name, 'AnchorPoint', 'LeftBottom', ...
        'BoxColor', 'w', 'FontSize', 32);
    imgs_for_tile{k} = mm_dataset{k}.im_label;
end

%%
% Visualize the whole dataset with labels
tiled = imtile(imgs_for_tile);
figure; imshow(tiled)
title('Manually labelled dataset of M&Ms', 'FontSize', 16);

%%
% Save dataset
% save('mm_dataset.mat', 'mm_dataset');

%% Sampling of feature vectors from dataset
% Sample the dataset using an isotropic Gaussian distribution,
% centered around the midpoint of the image.
%
% This will create some noise when samples are taken outside the M&M, but
% that is okay, it doesn't need to be perfect.
%
% We want feature vectors of about (64x1) to get a very robust classifier
% but using RGB gives three values per image pixel, so we can sample
% 21 pixels and build a feature vector of dimension (63x1).
%
% If we sample the same image 32 times (at different random locations),
% we then get a total of 32*32 = 1024 labelled reference vectors.

rng(42); % seed for reproducability
num_points = 21; % pixels sampled per image
rgb = 3; % 3-vector per pixel
num_samp_per_im = 32; % sample each im. 32 times
num_im = 32; % there are 32 im.s in the set

% Reference dataset variables
X = zeros([num_im*num_samp_per_im, num_points*rgb]); % 1024 feature vecs
y_name = {}; % Label strings
y_id = zeros([num_im*num_samp_per_im, 1]); % Label ids

z_frac = 1.5; % Gaussian quantile for the sampling distribution,
% the lower the number, the more noise.
% 2.58 -> about 1% misses,
% 1.96 -> about 5% misses,
% 1.64 -> about 10% misses,
% 1.50 -> about 12-14% misses, etc...
% E.g. norminv([0.005 .995]) -> [-/+ 2.58]

```



```

sample_idx = 1;      % Running counter

% BEGIN SAMPLING
for im_n = 1:num_im

    % Select image and compute settings
    samp_im = mm_dataset{im_n}.im;
    s = size(samp_im);

    % Central and max sampling coordinates
    y_ctr = s(1) / 2; y_max = s(1);
    x_ctr = s(2) / 2; x_max = s(2);

    % Sample the same image multiple times at different locations
    for sam_n = 1:num_samp_per_im

        % Sample locations for x-coordinate
        nrx = round(x_ctr + (x_ctr/z_frac)*randn([num_points, 1]));
        nrx(nrx <= 0) = 1;    % cap at min
        nrx(nrx > x_max) = x_max; % cap at max

        % Sample locations for y-coordinate
        nry = round(y_ctr + (y_ctr/z_frac)*randn([num_points, 1]));
        nry(nry <= 0) = 1;
        nry(nry > y_max) = y_max;

        % Obtain samples
        samples = zeros([num_points 3]); % Sample vector x 3 for RGB

        % sample the given number of different points in this image for
        % this feature vector
        for k = 1:num_points
            samples(k, :) = samp_im(nry(k), nrx(k), :); % RGB from im(x,y)
        end % end of sampling 21 points in same image for one feature vec

        % Organize so 21*R, 21*G, and 21*B
        feature_vec = reshape(samples, [], 1);

        % Store feature vector and labels
        X(sample_idx, :) = feature_vec'; % store feature vec
        y_id(sample_idx, 1) = mm_dataset{im_n}.label_id;
        y_name{end + 1} = mm_dataset{im_n}.label_name;

        % increment counter
        sample_idx = sample_idx + 1;

    end % end of sampling same image 32 times
end % end of sampling all images

%%
% Display last sampling points for demo
figure; imshow(samp_im); hold on;
plot(nrx, nry, 'r+');
hold off;
title('Bivariate Gaussian sampling points for an image');

```

```

%%
% Build query set (test set with labels, the labels are for accuracy test)
% Randomly select among the 32 images and perform a random sampling of the
% image until 100 images have been sampled.
% Store the labels to do accuracy measurement (backtest)

num_q_vecs = 100; % 100 query vectors
s_ims = randi([1 num_im], num_q_vecs, 1); % Sampled image ids

Xq = zeros([num_q_vecs, num_points*rgb]); % Feature vec's for query
yq_name = {}; % Label strings
yq_id = zeros([num_q_vecs, 1]); % Label ids

% Actually build the query set
rng(4242);

sample_idx = 1; % Running counter

for im_n = 1:num_q_vecs

    % Select random image and compute settings
    samp_im_d = mm_dataset{s_ims(im_n)};
    samp_im = samp_im_d.im; % get a random im
    s = size(samp_im);

    % Central and max sampling coordinates
    y_ctr = s(1) / 2; y_max = s(1);
    x_ctr = s(2) / 2; x_max = s(2);

    % For now, just do one sample per image
    for sam_n = 1:1

        % Sample locations for x-coordinate
        nrx = round(x_ctr + (x_ctr/z_frac)*randn([num_points, 1]));
        nrx(nrx <= 0) = 1; % cap at min
        nrx(nrx > x_max) = x_max; % cap at max

        % Sample locations for y-coordinate
        nry = round(y_ctr + (y_ctr/z_frac)*randn([num_points, 1]));
        nry(nry <= 0) = 1;
        nry(nry > y_max) = y_max;

        % Obtain samples
        samples = zeros([num_points 3]); % Sample vector x 3 for RGB

        % sample the given number of different points in this image for
        % this feature vector
        for k = 1:num_points
            samples(k, :) = samp_im(nry(k), nrx(k), :); % RGB from im(x,y)
        end % end of sampling 21 points in same image for one feature vec

        % Organize so 21*R, 21*G, and 21*B
        feature_vec = reshape(samples, [], 1);
    end
end

```

```

    % Store feature vector and labels
    Xq(sample_idx, :) = feature_vec'; % store feature vec
    yq_id(sample_idx, 1) = samp_im_d.label_id;
    yq_name{end + 1} = samp_im_d.label_name;

    % increment counter
    sample_idx = sample_idx + 1;

end % end of sampling same image 32 times
end % end of sampling all images

%%
% Display last sampling points for demo
figure; imshow(samp_im); hold on;
plot(nrx, nry, 'r+');
hold off;
title('Bivariate Gaussian sampling points for an image');

```

## 15.2 Appendix: Matlab code for flat file generation

```

%% Export the dataset to text files
% This is importable as a variable in a C header file
% In C, we load the reference (training) set as an array of 1024*63 ints,
% and we load the query (test) set as an array of 100*63 = 6300 ints.
% This is:
% - 258.0 kilobyte for the reference data, and
% - 25.2 kilobyte for the training set.
% First the reference/training data
data = X';          % to get the order I want! Full row, then next row, then...
len_data = size(data, 1)*size(data, 2);
fid_data = fopen('ref_vals.txt', 'wt');

% Write data to file as 1, 2, 3, 4, ...
for k = 1:len_data
    fprintf(fid_data, '%d', data(k)); % write value

    if k < len_data
        fprintf(fid_data, ',\n');      % write the comma
    end
end
fclose(fid_data);

% Then the corresponding labels (colour ids)
fid_labels = fopen('ref_labels.txt', 'wt');
labels = y_id;
len_labels = size(labels, 1)*size(labels, 2);
for k = 1:len_labels
    fprintf(fid_labels, '%d', labels(k));
    if k < len_labels
        fprintf(fid_labels, ',\n');
    end
end
fclose(fid_labels);

% The query/test data
data = Xq';          % to get the order I want! Full row, then next row, then...
len_data = size(data, 1)*size(data, 2);
fid_data = fopen('query_vals.txt', 'wt');
for k = 1:len_data
    fprintf(fid_data, '%d', data(k));
    if k < len_data
        fprintf(fid_data, ',\n');
    end
end
fclose(fid_data);

% Then the corresponding labels (colour ids)
fid_labels = fopen('query_labels.txt', 'wt');
labels = yq_id;
len_labels = size(labels, 1)*size(labels, 2);
for k = 1:len_labels
    fprintf(fid_labels, '%d', labels(k));
    if k < len_labels
        fprintf(fid_labels, ',\n');
    end
end
fclose(fid_labels);

```

## 15.3 Appendix: Matlab code for KNN algorithm development

### 15.3.1 Algorithm development for K = 1

```

%% Algorithm development
% Develop KNN for K=1

N = size(X, 1);      % Reference vectors in the dataset
fd = size(X, 2);     % Feature dimensions (63)
Nqv = size(Xq, 1);   % Query vectors in the dataset

% Find distance between one query vector and all N reference vectors
q = 1;               % look at query vector #1
qv = Xq(q, :);

% Compute distance to reference vector 1
rv = X(1, :);
sum_dist = 0;
for d = 1:fd
    sum_dist = sum_dist + (qv(d) - rv(d))^2;    % Sq. Euclidian dist
end

% Test that it gives same as squared norm of difference
assert (sum_dist == norm(qv-rv)^2)

% For all reference vectors -> vector of sq. Euclidian distances
% Nested loop
best_dist = Inf;
nearest_neighbour = 0;
for vecnum = 1:N
    sum_dist = 0;    % reset
    for d = 1:fd
        sum_dist = sum_dist + (qv(d) - X(vecnum, d))^2;
    end
    if (sum_dist < best_dist)    % New nearest neighbour
        best_dist = sum_dist;    % Store the sq-dist to nearest
        nearest_neighbour = vecnum; % Store the nearest so far
    end
end

% Find colour of nearest neighbour
predicted_colour = y_name(nearest_neighbour);
disp(['Nearest neighbour is ', predicted_colour{1}]);

% Check
actual_colour = yq_name(q);
disp(['The query vector is ', actual_colour{1}]);

```

15.3.2 Functional implementation of KNN,  $K = 1$ 

```

function [predictions] = predict_knn1(X, y_name, Xq)
% Make predictions using KNN with K = 1
% Janus Bo Andersen, December 2020

N = size(X, 1);      % Reference vectors in the dataset
fd = size(X, 2);      % Feature dimensions (63)
Nqv = size(Xq, 1);    % Query vectors in the dataset

predictions = {};

% Loop over all query vectors
for q = 1:Nqv
    qv = Xq(q, :);

    % For all reference vectors -> vector of sq. Euclidian distances
    % Nested loop
    best_dist = Inf;
    nearest_neighbour = 0;
    for vecnum = 1:N
        sum_dist = 0;    % reset
        for d = 1:fd      % loop over all features
            sum_dist = sum_dist + (qv(d) - X(vecnum, d))^2;
        end
        if (sum_dist < best_dist)      % New nearest neighbour
            best_dist = sum_dist;      % Store the sq-dist to nearest
            nearest_neighbour = vecnum; % Store the nearest so far
        end
    end

    % Find colour of nearest neighbour
    predicted_label = y_name(nearest_neighbour); % KNN K = 1, easy!
    predictions{q} = predicted_label{1};        % Remove cell wrapper
end % end of loop over all q

end

```

15.3.3 Verification of  $K = 1$  KNN against Matlab built-in

```

%% Test dataset using Matlab's KNN
% Test this dataset using Matlab's KNN, with K = 1

Mdl = fitcknn(X,y_name,'NumNeighbors', 1);

predictions = Mdl.predict(Xq); % Predict the query data based on training
actual = yq_name';           % the labels we saved from random samples

correct = 0;
for k=1:num_q_vecs
    if strcmp(predictions{k}, actual{k})
        correct = correct + 1;
    end
end

accuracy = correct / num_q_vecs;
disp(['Accuracy: ', num2str(round(accuracy * 100, 2)), '%.']);

%% Test own implementation and compare to Matlab
% Run for all 100 test vectors
own_knn1_predict = predict_knn1(X, y_name, Xq)';

% compare to Matlabs predictions
Mdl = fitcknn(X,y_name,'NumNeighbors', 1);
matlab_knn1_predict = Mdl.predict(Xq);

for k = 1:100
    assert (strcmp(own_knn1_predict{k}, matlab_knn1_predict{k}) == true)
end

% ALL GOOD!

% find the misses for our own
for k = 1:100
    own_knn1_misses(k) = ~strcmp(own_knn1_predict{k}, yq_name{k});
end

own_knn1_accuracy = 1 - sum(own_knn1_misses) / 100;

```

15.3.4 Algorithm development for general  $K$ 

```

%% Algorithm development
% Develop KNN for any K

K = 3;

q = 1;                                % look at query vector #1
qv = Xq(q, :);

d2 = zeros([N 1]);                     % Store sq. Euclidian dists
for vecnum = 1:N
    sum_dist = 0;                       % reset sum
    for d = 1:fd
        sum_dist = sum_dist + (qv(d) - X(vecnum, d))^2;
    end
    d2(vecnum) = sum_dist;              % Store the distance
end

% Find K nearest neighbours by sorting and searching
sort_dists = sort(d2, 'ascend');
K_min_dists = sort_dists(1:K);
for k = 1:K
    K_min_idx(k) = find(d2 == K_min_dists(k));
    K_min_clr_ids(k) = y_id(K_min_idx(k));
end

% Voting
knn = mode(K_min_clr_ids);              % This might be a bit tricky to impl.

% Translate from id to colour name, e.g. 1 -> 'blue'
clr_names = {'blue', 'yellow', 'brown', 'red', 'green'};

% Find colour of nearest neighbour
predicted_colour = clr_names(knn);
disp(['Nearest neighbour classification (K=', num2str(K), ') is ', ...
    predicted_colour{1}]);

% Check
actual_colour = yq_name(q);
disp(['The query vector is ', actual_colour{1}]);

```



15.3.5 Functional implementation of KNN, general  $K$ 

```

%%
%
function [predictions] = predict_knnK(X, y_id, Xq, K)
% Make predictions using our own KNN model, for any choice of K
% Janus Bo Andersen, December 2020

% Translate from id to colour name, e.g. 1 -> 'blue'
clr_names = {'blue', 'yellow', 'brown', 'red', 'green'};

N = size(X, 1);      % Reference vectors in the dataset
fd = size(X, 2);      % Feature dimensions (63)
Nqv = size(Xq, 1);   % Query vectors in the dataset

predictions = {};

for q = 1:Nqv
    qv = Xq(q, :);      % Choose query vector

    d2 = zeros([N 1]);   % Store sq. Euclidian dists
    for vecnum = 1:N
        sum_dist = 0;    % reset sum
        for d = 1:fd
            sum_dist = sum_dist + (qv(d) - X(vecnum, d))^2;
        end
        d2(vecnum) = sum_dist; % Store the distance
    end

    % Find K nearest neighbours by sorting and searching
    sort_dists = sort(d2, 'ascend');
    K_min_dists = sort_dists(1:K);
    for k = 1:K
        K_min_idx(k) = find(d2 == K_min_dists(k));
        K_min_clr_ids(k) = y_id(K_min_idx(k));
    end

    % Voting among the K nearest neighbours
    knn = mode(K_min_clr_ids); % This might be tricky to impl. in C++

    % Find colour of nearest neighbour
    predicted_colour = clr_names(knn);
    predictions{q} = predicted_colour{1};
end % end for q

end

```

## 15.3.6 Verification of general KNN against Matlab built-in

```

%% Test against Matlab
%

own_knn3_predict = predict_knnK(X, y_id, Xq, 3);

% find the misses for our own
for k = 1:100
    own_knn3_misses(k) = ~strcmp(own_knn3_predict{k}, yq_name{k});
end

own_knn3_accuracy = 1 - sum(own_knn3_misses) / 100;

% Train matlab model
Mdl = fitcknn(X,y_name,'NumNeighbors', 3);
matlab_knn3_predict = Mdl.predict(Xq);

% find the misses for Matlab
for k = 1:100
    matlab_knn3_misses(k) = ~strcmp(matlab_knn3_predict{k}, yq_name{k});
end

% Compare predictions
for k = 1:100
    assert (strcmp(own_knn3_predict{k}, matlab_knn3_predict{k}) == true)
end

```

## 15.4 Appendix: C++ code for implementation of KNN

### 15.4.1 File: main.cpp

```

/*
 * Advanced Digital Design (E5ADD) 2020
 * Implementation of k-Nearest Neighbours on Zynq-7000
 * M&Ms colour classification system
 * Oct-Dec 2020
 *
 * Files:
 * knn.h           HW accelerator prototype and SDS interfaces
 * knn.cpp         HW accelerator implementation
 * knn_support.h   SW KNN algorithm, voting function and scoring
 * dataset.h       Contains metadata and loads the flat files in data/
 * ref_vals.txt    1024 reference vectors (training) with 63 features
 * ref_labels.txt  Labels for each of the 1024 reference vectors
 * query_vals.txt  100 query vectors (test) with 63 features
 * query_labels.txt Labels for each of the 100 query vectors
 */

#include <knn.h>
#include <iostream>
#include <cstring>
#include <stdlib.h>
#include <climits>
#include <stdio.h>
#include "sds_utils.h"
#include <vector>
#include <algorithm>
#include <stdint.h>
#include "knn_support.h"
#include "dataset.h"

int main(int argc, char** argv)
{
    // Get K from user
    int knn k;
    std::cout << "Perform k-Nearest Neighbours with k = ? Please enter a
positive integer:" << std::endl;
    std::cin >> knn k;
    std::cout << "Starting KNN with k=" << knn k << "." << std::endl;

    // Heap allocation
    uint8 t* ref vecs = (uint8 t*) sds alloc(sizeof(uint8 t) * DATA SIZE
* DATA DIM); // Reference (train) set
    uint8 t* query = (uint8 t*) sds alloc(sizeof(uint8 t) *
QUERY_SIZE * DATA DIM); // Query (test) set
    unsigned int* d2 hw = (unsigned int*) sds alloc(sizeof(unsigned
int)*DATA SIZE); // 1024 return vals

    // Heap allocation for PS algo.
    unsigned int* d2 sw = (unsigned int*) malloc(sizeof(unsigned
int)*DATA SIZE);

    // Ensure OK memory alloc.
    if((ref_vecs == NULL) || (query == NULL) || (d2_hw == NULL) || (d2_sw
== NULL)){
        std::cout << "Could not allocate on the heap." << std::endl;
        return -1;
    }
}

```

```

// Load datasets onto heap to ensure similar access from both HW and SW
for (int i = 0; i < DATA_SIZE*DATA_DIM; i++) {
    ref_vecs[i] = mms::reference_vectors[i];
}

for (int i = 0; i < QUERY_SIZE*DATA_DIM; i++) {
    query[i] = mms::query_vectors[i];
}

// Use the profiling method from SDSoc guide
sds_utils::perf_counter hw_ctr, sw_ctr;

// Classify all 100 query vectors using the hardware accelerator
std::vector<int> hw_results;
uint8_t* q_ptr = query;

// profile HW algo. only during HW runtime
for (int q = 0; q < QUERY_SIZE*DATA_DIM; q += DATA_DIM) {

    q_ptr = query + q; // advance pointer to next query vector
    hw_ctr.start();

    // Compute KNN with hardware accelerator for q'th query vector
    #pragma SDS data mem attribute(ref_vecs:PHYSICAL CONTIGUOUS, q_ptr:PHYSICAL CONTIGUOUS,
d2_hw:PHYSICAL CONTIGUOUS)
    knn_hw(ref_vecs, q_ptr, d2_hw);
    //knn::knn_hw(mms::reference_vectors, mms::query_vectors + q, d2, DATA_SIZE, DATA_DIM);
    hw_ctr.stop();

    // Classify this vector based on squared distances in d2
    auto cls_idx = knn::find_nearest_neighbours(knn_k, d2_hw, mms::reference_labels,
DATA_SIZE, mms::class_ids);
    hw_results.push_back(cls_idx); // Store, these are array indices, not the classes
themselves
}

// Compare results and rate
std::cout << "*** Hardware KNN results ***" << std::endl;
auto hw_accuracy = knn::knn_rate(hw_results, mms::query_labels, QUERY_SIZE,
mms::class_ids);
std::cout << "Accuracy: " << hw_accuracy << std::endl;

// Classify all 100 query vectors using the software implementation on Arm Cortex-A9
std::vector<int> sw_results;

// profile SW algo. only during equivalent runtime to the HW
for (int q = 0; q < QUERY_SIZE*DATA_DIM; q += DATA_DIM) {

    q_ptr = query + q; // advance pointer to next query vector
    sw_ctr.start();

    // Compute KNN in software for q'th query vector
    knn::knn_sw(ref_vecs, q_ptr, d2_sw, DATA_SIZE, DATA_DIM);

    sw_ctr.stop();
}

```

```

        // Classify this vector based on squared distances in d2
        auto cls_idx = knn::find_nearest_neighbours(knn_k, d2_sw, mms::reference_labels,
DATA_SIZE, mms::class_ids);
        sw_results.push_back(cls_idx); // Store, these are array indices, not the classes
        themselves
    }

    // Compare results and rate
    std::cout << "*** Software KNN results ***" << std::endl;
    auto sw_accuracy = knn::knn_rate(sw_results, mms::query_labels, QUERY_SIZE,
mms::class_ids);
    std::cout << "Accuracy: " << sw_accuracy << std::endl;

    // Verify HW algorithm: Confirm that HW and SW get same results, sqdiff must be zero
    unsigned long sq_diff = 0;
    for(int i = 0; i < DATA_SIZE; i++) {
        sq_diff += SQ(d2_sw[i] - d2_hw[i]);
    }

    std::cout << "*** Verification report ***" << std::endl;
    if(sq_diff != 0) {
        std::cout << "BAD: HW and SW Euclidian distances do not match." << std::endl;
    } else {
        std::cout << "OK: HW and SW Euclidian distances match." << std::endl;
    }

    std::cout << "*** Performance report ***" << std::endl;
    uint64_t sw_cycles = sw_ctr.avg_cpu_cycles();
    uint64_t hw_cycles = hw_ctr.avg_cpu_cycles();

    double speedup = (double) sw_cycles / (double) hw_cycles;

    std::cout << "Avg. CPU cycles running KNN in software: " << sw_cycles << std::endl;
    std::cout << "Avg. CPU cycles running KNN in hardware: " << hw_cycles << std::endl;
    std::cout << "Speed-up: " << speedup << std::endl;

    // Free heap
    sds_free(ref_vecs);
    sds_free(query);
    sds_free(d2_hw);
    free(d2_sw);

    return 0;
}

```

## 15.4.2 File: knn.h

```

#ifndef KNN_H
#define KNN_H

#include <stdint.h>

#define DATA_DIM 63 // features in a feature vector
#define DATA_SIZE 1024 // size of the reference (training) set

// copy will infer DMA (FastDMA) transfer to PL (from contiguous mem area!), and specify copy
// sizes
// data access is strictly required to be sequential for speed-up
#pragma SDS data copy(ref_vecs[0:DATA_SIZE*DATA_DIM], query[0:DATA_DIM], d2[0:DATA_SIZE])
#pragma SDS data access pattern(ref_vecs:SEQUENTIAL, query:SEQUENTIAL, d2:SEQUENTIAL)
#pragma SDS data mem attribute(ref_vecs:PHYSICAL CONTIGUOUS, query:PHYSICAL CONTIGUOUS,
d2:PHYSICAL CONTIGUOUS)
void knn_hw(uint8_t* ref_vecs, uint8_t* query, unsigned int* d2);

#endif

```

## 15.4.3 File: knn.cpp

```

#include "knn.h"

#define SQ(x) ((x)*(x))

void knn_hw(
    uint8_t *ref_vecs, // Reference vectors (training data)
    uint8_t *query,    // Query vector (test vector)
    unsigned int *d2    // Sq. Euclidian distances return value
)
{
    // BRAM block for local memory
    uint8_t query_local[DATA_DIM];

    // Partition the local BRAM for more concurrency. A high factor puts data in FFs.
    #pragma HLS ARRAY_PARTITION variable=query_local cyclic factor=32 dim=1

    // Running sum
    unsigned int sum_dist = 0;

    // Get entire query vector (from DMA) to local BRAM to be read multiple times
    get_query: for(int i = 0; i < DATA_DIM; i++){
        #pragma HLS PIPELINE
        query_local[i] = query[i];
    }

    // Perfect nested loop
    for_each_ref_vec: for(int i = 0; i < DATA_SIZE; i++) {
        compute_dist: for(int j = 0; j < DATA_DIM; j++) {
            #pragma HLS PIPELINE
            #pragma HLS UNROLL factor=32

            // MAC operation to compute sq. Euclidian distance, getting ref_vecs sequentially
            sum_dist += SQ(query_local[j] - ref_vecs[i*DATA_DIM + j]);

            if(j == DATA_DIM-1) {
                d2[i] = sum_dist; // send final sq. dist. to i-th ref.vec.
                sum_dist = 0;    // reset
            }
        }
    }
}

```

## 15.4.4 File: knn\_support.h

```

/*****
 * Finding K Nearest Neighbours Algorithm
 * Created by Janus Bo Andersen, Dec 2020
 *
 * find nearest neighbours:
 * In array of squared distances, "d2"
 * with corresponding labels in "labels",
 * both of length "size",
 * the algo finds "K" smallest distances,
 * and by counting votes for each class
 * based on the classes from "all classes",
 * the most frequent (mode) is picked.
 * if it is a tie, the class for the nearest
 * neighbour is chosen.
 *
 * knn_sw: Compute sq. Euclidian distances
 *
 * knn_rate: Rate the accuracy of the classifier
 *****/

#ifdef KNN_KNN_SUPPORT_H
#define KNN_KNN_SUPPORT_H
#define SQ(x) ((x)*(x))

#include <vector>
#include <algorithm>
#include <stdint>

namespace knn {
    int find_nearest_neighbours(const int K,
                                const unsigned int* distances,
                                const uint8_t* labels,
                                const int size,
                                const std::vector<uint8_t>& all_classes
    ) {

        // Store distance-label pairs, we sort the first and get label from the second
        std::vector<std::pair<int, uint8_t>> dl_pairs;
        for (int i = 0; i < size; i++) {
            dl_pairs.push_back(std::make_pair(distances[i], labels[i]));
        }

        // Sort the pairs in place from lowest to highest
        std::sort(dl_pairs.begin(), dl_pairs.end());

        // For the K lowest distances, extract the class label
        std::vector<int> k_classes;
        for (int k = 0; k < K; k++) {
            k_classes.push_back(dl_pairs[k].second);    // grab the class label
        }

        // Count votes for each class in the original class list
        std::vector<int> k_count;
        for(auto& cls: all_classes) {
            k_count.push_back(std::count(k_classes.begin(), k_classes.end(), cls));
        }
    }
}

```

```

// Find the most frequent and translate it into a class index
auto k_maxvote_itr = std::max_element(k_count.begin(), k_count.end());
auto k_maxvote_idx = std::distance(k_count.begin(), k_maxvote_itr);

// Returns the index for the most popular class in all_classes
return (int) k_maxvote_idx;
}

void knn_sw(const uint8_t* ref_vecs, // "size" reference vectors each of length "dim"
            const uint8_t* query,   // 1 query vector of length "dim"
            unsigned int* d2,        // allocated space to store sq. Euclid. dists.
            int size,                // number of reference vectors
            int dim)                 // number of features in a vector
{
    unsigned long sum_dist;

    // Nested loop to go through all reference vectors
    for (int i = 0; i < size; i++) {
        sum_dist = 0;

        // MAC operation to go through all vector components
        for (int j = 0; j < dim; j++) {
            sum_dist += SQ(query[j] - ref_vecs[i*dim + j]);
        }

        // Store the squared Euclidian distance
        d2[i] = sum_dist;
    }
}

float knn_rate(const std::vector<int>& results, // array indices
               const uint8_t* query_labels,
               const int query_size,
               const std::vector<uint8_t>& class_ids
               ) {
    // Compare results and rate
    int predict;
    int truth;
    int missed = 0;
    for (int q = 0; q < query_size; q++) {
        predict = class_ids[results[q]];
        truth = (int) query_labels[q];

        if (predict != truth) {
            missed++;
            std::cout << "Misclassification of obs. " << q+1 << " out of " << query_size
            << "." <<
                " Pred.: " << predict << ", True: " << truth << std::endl;
        }
    }

    return 1 - missed/(float) query_size;
}

#endif //KNN_KNN_SUPPORT_H

```



## 15.4.5 File: dataset.h

```
//
// Created by Janus Bo Andersen, Dec 2020
//

#ifndef KNN_DATASET_H
#define KNN_DATASET_H

#include <vector>
#include <string>
#include <stdint>

#define REF_SIZE 1024
#define FEATURE_DIM 63
#define QUERY_SIZE 100

namespace mms {
    const uint8_t reference_vectors[REF_SIZE * FEATURE_DIM] = {
        #include "data/ref_vals.txt"
    };

    const uint8_t reference_labels[REF_SIZE] = {
        #include "data/ref_labels.txt"
    };

    const uint8_t query_vectors[QUERY_SIZE * FEATURE_DIM] = {
        #include "data/query_vals.txt"
    };

    const uint8_t query_labels[REF_SIZE] = {
        #include "data/query_labels.txt"
    };

    const std::vector<uint8_t> class_ids {1, 2, 3, 4, 5};
    const std::vector<std::string> class_names {"blue", "yellow", "brown", "red", "green"};
}

#endif //KNN_DATASET_H
```

## 15.4.6 File: sds\_util.h

```
/*
 * https://github.com/Xilinx/SDSoC_Examples/blob/master/libs/sds_utils/sds_utils.h
 */

#ifndef SDS_UTILS_H
#define SDS_UTILS_H
#include <stdint.h>
#include "sds_lib.h"
namespace sds_utils {
    class perf_counter
    {
    private:
        uint64_t tot, cnt, calls;

    public:
        perf_counter() : tot(0), cnt(0), calls(0) {};
        inline void reset() { tot = cnt = calls = 0; }
        inline void start() { cnt = sds_clock_counter(); calls++; }
        inline void stop() { tot += (sds_clock_counter() - cnt); }
        inline uint64_t avg_cpu_cycles() {return (tot / calls); }
    };
}

#endif
```