Technische Universität Berlin

Applied Embedded Systems Project

Project Paper Summer Semester 2015

Hardware accelerator for kNN matching

Authors:

Teodora Anitoaei (368014)

Sonali Deo (368020)

Varun Gowtham (368053)

Sagar Sharma (368058)

Mentors:

Ahmed Elhossini

Kawa Haji Mahmoud

August 7, 2015

August 7, 2015

Contents

1 Introduction **2**

2 Image matching **3**

3 Parallelization of matching and Test Setup **5**

4 Implementation on GPU and Intel chips **9**

4.1 Non-parallel single thread 9

4.2 Parallelizing first for loop; launching NK1 threads 9

4.3 Parallelizing first and second for loop; launching NK1*NK2 threads 10

5 Implementation on Zedboard **11**

5.1 Preliminaries 11

5.2 Design 11

5.3 Conversion from C to VHDL code 14

5.4 Creation of custom IP core 17

5.5 Steps for IP core creation 17

6 Conclusion **17**

Abstract

With increased number of sensors collecting massive amounts of data, it is more evident that processing of data takes a key role in interpreting the aggregated data. Particularly in the field of image processing it is required that data may be processed independently for the purpose of an algorithm. One such algorithm used for recognition of images between a train image and query image is use of k-Nearest Neighbors which by brute force compares the key points generated of the images to ascertain the extent of matching between the images. This process is time consuming. This project report concentrates on parallelizing the matching operation.

1 Introduction

During the recent advances in technology ability to accumulate data has exponentially increased. Research is concentrating on ways to process the data for useful interpretation. In this report we introduce a problem where parallelization of computation of data is at prime importance. We introduce a typical problem of image recognition between two images with k-Nearest Neighbors (kNN) matching running under the facility of KAZE features extracted from the given images.

In data mining applications, one of the useful algorithms for classification is the kNN algorithm. The kNN search has a wide usage in many research and industrial domains like 3-dimensional object rendering, content-based image retrieval, statistics, biology (gene classification), etc. In spite of some improvements in the last decades, the computation time required by the kNN search remains the bottleneck for kNN classification, especially in high dimensional spaces. This problem has created the necessity of the parallel kNN on commodity hardware. GPU is one of the low cost high performance solutions for parallelizing the kNN classifier.

In our problem images are read from memory to extract features. Feature extraction is performed by using KAZE feature in a multi-scale 2D feature detection and description algorithm. It describes 2D features in a non-linear scale space by means of non-linear diffusion filtering. This method of feature extraction helps in efficient feature extraction as compared to algorithms using Gaussian blurring. KAZE feature extraction defines clear features, thus enabling us to retrieve key points for matching and descriptors for each key point obtained.

Therefore, the steps for matching between the training image and query image include feature detection, description matching. Feature detection involves defining key points in the images. Working on these key points, feature description uses 1st order derivatives and other operations to result in scale and rotation invariant descriptors. This process is applied on images obtained from non-linear scale space (rescaling the original image), and the resultant descriptors (vectors) are matched key point by key point to see if the images under observation using kNN matching and give an extent or estimate to which the images match each other.

This report concentrates on explaining the implementation of the project for matching the images. The approaches can be viewed as a non-parallel single thread approach and parallel multi-threaded approach which are described in the following sections.

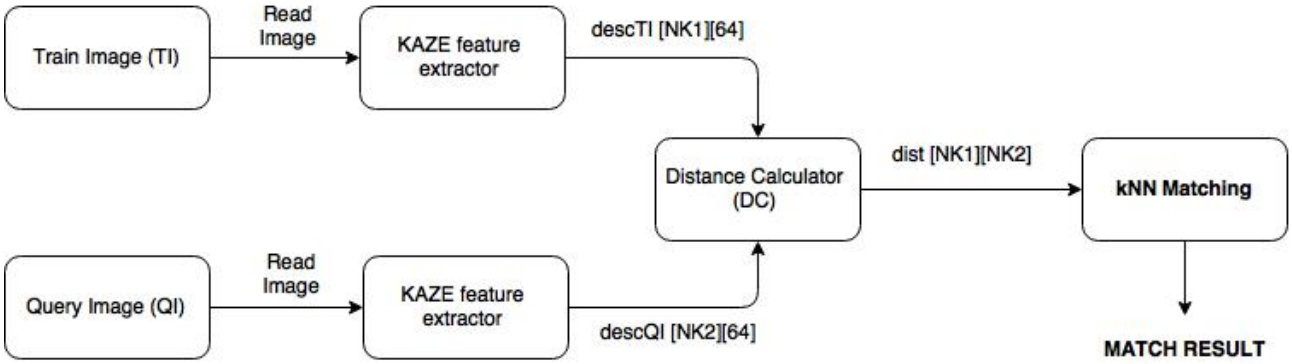

Figure 1: KAZE kNN matching procedure

2 Image matching

This section will give a brief overview of the procedure followed in matching a Train Image (TI) and Query Image (QI). TI is an image which will be used to extract useful reference key points of the image. QI is an image which is required to be compared to the TI to estimate the extent to which it matches with the TI. This is done by k-NN matching, which compares the distances between the descriptors of key points found in TI with descriptors of key points found in QI to locate matching key points between QI and TI. Based on the number of matches found between TI and QI, we can say the extent to which both TI and QI match each other.

For the implementation we will consider the matching to be performed with k=2 i.e k-NN matching locates two most nearest neighbors to classify if a given key point in TI matches to a key point in QI based on thresholding criteria. Fig. 1 shows a block diagram of the procedure for the k-NN matching. QI and TI are read from memory and fed into the KAZE feature extractor. The feature extractor is a program block which extracts key points using KAZE features and also gives descriptors for each key point located in the given image. Therefore TI would produce NK1 number of key points and QI would produce NK2 number of key points. Each descriptor representing a key point is an array of length 64. After the stage of feature extraction we get two 2-dimensional arrays namely descTI having NK1 rows and 64 columns and descQI having NK2 rows and 65 columns. Each value of the elements in descriptors are floating point values.

Next stage in the flow diagram is Distance calculator (DC). The DC is a simple module which calculates n-space Euclidean distance between a key point in TI to a key point in QI where n is 64. Therefore after the completion of calculating all distances, the output from the block will be 2-dimensional vector of distance namely dist which has NK1 rows and NK2 columns (dist[NK1][NK2]). The elements in dist are floating point value.

$$dist[x][y] = \sqrt{(descTI[x][0] - descQI[y][0])^2 + ... + (descTI[x][63] - descQI[y][63])^2} \quad (1)$$

Eq.-1 shows a mathematical formula to obtain the euclidean distance; dist[x][y] contains the distance between "x" key point of TI to "y" key point of QI.

Next stage of the procedure is k-NN matching. There are two steps, first iterate through the elements of each row of dist to obtain the two most minimum values to obtain the knndist which is a 2 dimensional vector having NK1 rows and 2 columns (knndist[NK1][2]). It is important to note that the first column (knndist[x][0]) of a given row x contains the most minimum value and the second column (knndist[x][1]) contains the next most minimum value. Also knnloc records the key point number in QI where the most minimum value (knn[x][0]) was located. Therefore in the next stage if a match criteria is satisfied the index of knnloc will represent a key point in TI and the corresponding element will represent the matched key point in QI. Second step of the process is to apply a thresholding condition to determine a match. This is done by comparing the values of each row of knndist against a threshold (in our case it is 0.8). In particular, in our implementation, we check if (knndist[x][0] < 0.8 * knndist[x][1]), if the condition turns out to be true then a match criteria is satisfied and the keypoints are considered to be matched (index of knnloc matched to value of corresponding element).

Further the matched key points between TI and QI are used to visualize and evaluate the extent of match between the images.

```
float knndist[NK1][2]; // two most minimum distances for each key point in TI with each
                       // key point in QI
int knnloc[NK1]; // key point of QI where most minimum distance was found
int nidx[2]; // utility variables
double bufvalue =0;
float bufvalue_ref;
bool switchs = false;
for(int i=0; i<NK1; i++){ // loop iterating through each key point in TI
// *************** First for loop ************************************
  bufvalue = 0;
  knndist[i][0] = (double) FLT_MAX; // initialize with MAX value
  knndist[i][1] = (double) FLT_MAX;
  nidx[0] = -1; // initialize with negative value
  nidx[1] = -1;
  switchs = false;
    for(int j=0; j<NK2; j++){
// *************** Second for loop ************************************
      bufvalue = 0.0;
      for(int k=0; k<64; k++){ // loop iterating to calculate the euclidean distance
```

```
// *************** Third for loop ***************************************
        float t = descTI[i][k] - descQI[j][k];
        bufvalue += (float) t*t;
      }
      bufvalue_ref = (float)sqrt(bufvalue); // Euclidean distance between key point "i"
                                            //  of TI and key point "j" of QI
      if(bufvalue_ref < knndist[i][0] ){ // section to find two most minimum distances
        if(switchs) {
          nidx[1] = nidx[0];
          knndist[i][1] = knndist[i][0];
        }
          nidx[0] = j; // save the location of the key point
                       // of QI where most minimum distance was found
          knndist[i][0] = bufvalue_ref;
          switchs = true;
      }
    }
  }
        knnloc[i] = -1;
        if( nidx[0] > 0){
          knnloc[i] = nidx[0]; // update the minimum distance point location to knnloc
}

for(int i=0; i < NK1; i++){
  if (knnloc[i] > 0)
    if(knndist[i][0] < (0.8f*knndist[i][1])) { // check threshold, satisfied ?
                                               // add as a match else discard
      // Key point "i" of TI matches with
      //key point of QI indexed by value in knnloc[i]
    }
}
```

The above pseudo code represents the block diagrams represented by Fig.-1. We can see that the C-code accepts input descTI and descQI to process them and output matched key points between TI and QI.

3 Parallelization of matching and Test Setup

Pseudo code shown in the previous section hints that we can parallelize few aspects in order to achieve faster execution times. It can be seen that we have an opportunity to exploit parallelism in the step of calculation of euclidean distance. The code for calculating the euclidean distance runs for NK1*NK2 iterations with each iteration taking a pair of descriptors, one from descTI and descQI.

Again each iteration writes the result to a unique memory location. Clearly calculation of euclidean distance is independent and can be exploited to parallelize and speed up the matching time taken. We have several opportunities for parallelization, namely:

1. **Unroll first for loop:** Launching NK1 number of threads in parallel, each thread representing operation on behalf of a key point in TI performing the operations performed by the first for loop in a single iteration. Each thread generates NK2 number of distances and calculates two most minimum distances for a given key point in TI.

2. **Unroll both first and second for loops:** Launching NK1*NK2 number of threads in parallel, with each thread calculating distances for a given pair of key points from QI and TI.

3. **At calculation of euclidean distance:** This approach speeds up the execution by parallel execution of operations in the third for loop. Each thread calculating the difference between corresponding elements of descriptors, squaring the difference and wait for all threads to complete. Add all the squared differences from all threads and perform a square root operation on the whole result to obtain the euclidean distance.

The project aims to implement and test the parallel methods suggested above using Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA) and compare it with single thread implementations. We have used GPU to demonstrate applications of approaches 1. and 2. while using FPGA to demonstrate approach 3. Approach 1. could also be taken using the Intel Thread Building Blocks (TBB) [1] used by OpenCV [2] in the form of "parallel_for_" application programming interface available at user space to use this feature on intel processors.

For the experiment we have chosen 6 images. The first image shown in Fig.-2 is considered as train image and the images shown in Fig-3 to Fig.-7 are used as query images. It can be noted that the key points located by KAZE features for the given images are 6187, 6325, 5177, 3481, 3056 and 2751 respectively for the images. The procedure for kNN matching would be to locate matching key points between a train image and a query image. Based on the ratio of number of key points in train image to matched inlier key points we get an estimate on the extent to which train image is similar to query image. Apart from the parameter of inlier key-point ratio, time required for the operation of matching is important understand the effects of parallelism on the matching.

For our experiments we have used NVIDIA GeForce 840M available as a discrete GPU on a personal computer. The GPU is programmed using NVIDIA CUDA with NVIDIA driver version 346. The FPGA implementations are carried out using ZedBoard available from Avnet [3].

Figure 2: Train Image - Image1

Figure 3: Query Image - Image2

Figure 4: Query Image - Image3

Figure 5: Query Image - Image4

Figure 6: Query Image - Image5

Figure 7: Query Image - Image6

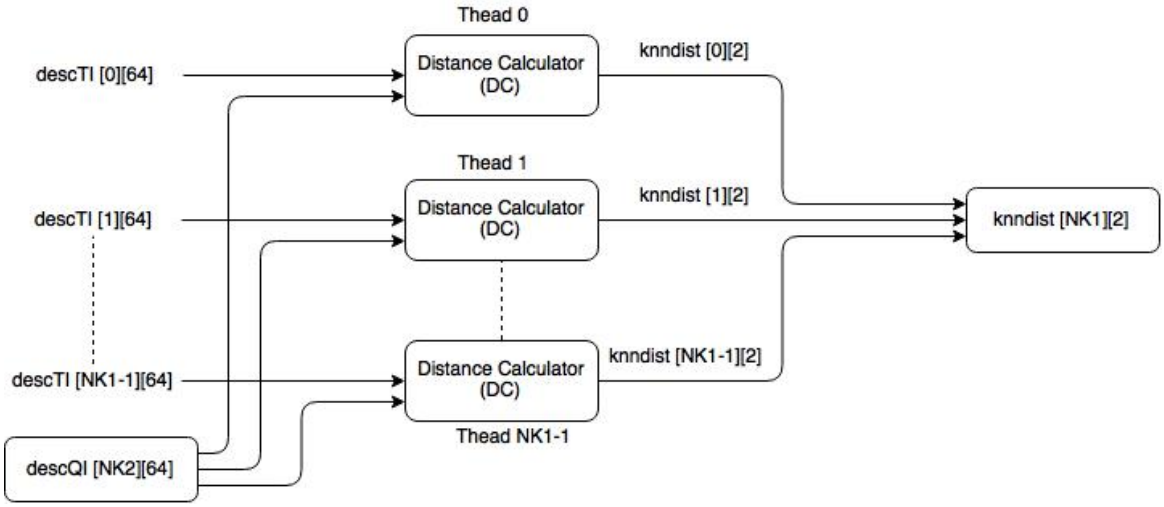

Figure 8: Parallelizing at first for loop, NK1 threads

4 Implementation on GPU and Intel chips

4.1 Non-parallel single thread

The pseudo code that has been shown in the previous section was implemented and tested. Table-1 shows the statistics for the experiment. Here the statistics are taken from image in Fig.-2 as train image and rest of the images as query images. It can be seen that the time taken for matching key points between the images is quite long shown in the last row of the table. The extent to which a query image matches a train image is given by the inliers key point ratio. Higher the value higher the match between the images.

Label	Image 2	Image 3	Image 4	Image 5	Image 6
Matches	4153	2809	1548	942	618
Inliers	3565	2313	1071	508	129
Inliers Match ratio	85.84	82.34	69.18	53.92	20.87
Inliers key point ratio	56.98	40.70	22.15	10.99	2.88
Time (ms)	9761.2	8072.25	5353.13	4725.5	4661.16

Table 1: Non parallel single thread implementation

4.2 Parallelizing first for loop; launching NK1 threads

Fig.-8 shows a block diagram representing the data flow and the thread operation when the first for loop in the pseudo code is parallelized. Each thread launched in parallel from GPU to calculate two most minimum distances with respect to all key points in QI given a single key point in TI. Each thread is free to read the required key point descriptor stored in the memory. The output of the thread is written into a location pointed by the thread number into the knndist variable. This collective result after the CUDA kernel exits, is compared with thresholding to decide if there was a match according

to the pseudo discussed earlier. Table.-2 shows the statistics for the given case.

Label	Image 2	Image 3	Image 4	Image 5	Image 6
Matches	4127	2784	1527	920	605
Inliers	3546	2297	1056	490	126
Inliers Match ratio	85.92	82.5	69.15	53.26	20.82
Inliers key point ratio	56.68	40.42	21.84	10.6	2.81
Time (ms)	1201.47	868.17	575	502.71	460.6

Table 2: Statistics for GPU NK1 threads implementation

OpenCV [2] also has an implementation which used Intel Thread Building Blocks (TBB) capable of parallelizing NK1 threads discussed in this section. This is done using the "parallel_for_" application programming interface to allocate data required and launch NK1 threads to execute in parallel as shown in Fig-8. It can be seen that the time taken for matching between the images is less in using the OpenCV TBB implementation.

Label	Image 2	Image 3	Image 4	Image 5	Image 6
Matches	3386	2222	1149	552	265
Inliers	3328	2016	966	430	106
Inliers Match ratio	90.28	90.72	84.07	77.89	40
Inliers key point ratio	53.19	35.48	19.98	9.3	2.37
Time (ms)	924	793.16	597.46	553	539

Table 3: Statistics for OpenCV on Intel microprocessor with NK1 threads implementation

4.3 Parallelizing first and second for loop; launching NK1*NK2 threads

This implementation focuses on parallelizing the operation of matching with launching NK1*NK2 threads with each thread processing the distance between a pair of descriptors, one taken from the TI and other taken from the QI. Fig.-9 shows the distribution of work among the threads. It can be seen that the time of execution has reduced compared to the NK1 threads implementation using the GPU.

Label	Image 2	Image 3	Image 4	Image 5	Image 6
Matches	4127	2784	1527	920	605
Inliers	3546	2297	1056	490	126
Inliers Match ratio	85.92	82.5	69.15	53.26	20.82
Inliers key point ratio	56.68	40.42	21.84	10.6	2.81
Time (ms)	1072.01	786.98	412.5	352.77	353.87

Table 4: Statistics for GPU NK1 threads implementation

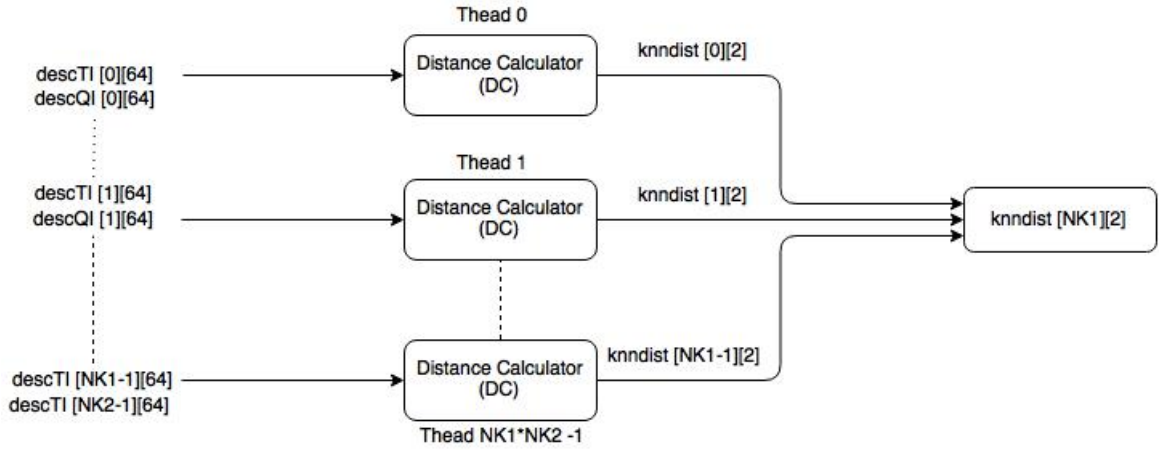

Figure 9: Parallelizing at first and second for loop, NK1*NK2 threads

5 Implementation on Zedboard

5.1 Preliminaries

We started with experimenting file read and write from SD card through Zedboard. This involved simple file reading and dumping. As we are feeding the train and query descriptors to the board using an SD card, this was a viable step to start with.

Following this, we tried to convert a text file containing keypoint values into one dimensional array of size 64. The text file has keypoints separated by comma and newline after every 64 values indicating new row where number of rows are variable. At a time, 64 values are stored in a buffer using fgets(), converted to array of type int32_t using strtol() and then used for computation.

Timing specification was another aspect of the code we worked on, so as to indicate how much time the hardware and software implementation of kNN matching takes. Our overall implementation compares the time taken to match NK1 key points and NK2 key points from TI and QI respectively. We chose to use the inherent Triple Timer Counters (TTC) provided by the processors in Zynq for counting the clocks elapsed in performing computations for kNN matching using hardware and software shown in Fig.-10. TTC can be visualized from the following diagram depicting its components, registers and I/O ports.

5.2 Design

The C code (derived from C++ code referencing OpenCV functions) for kNN matching, with k=2 had two sections of the code containing loops which were the most time consuming parts of the algorithm. One loop produced distances between all the keypoints in a row (i.e. 64), squared and added the distances and outside the loop, computed the square root of overall distance between two rows (one from train and another from query descriptor). The code is depicted below:

```
loop1: for(int k=0; k<64; k++)
```

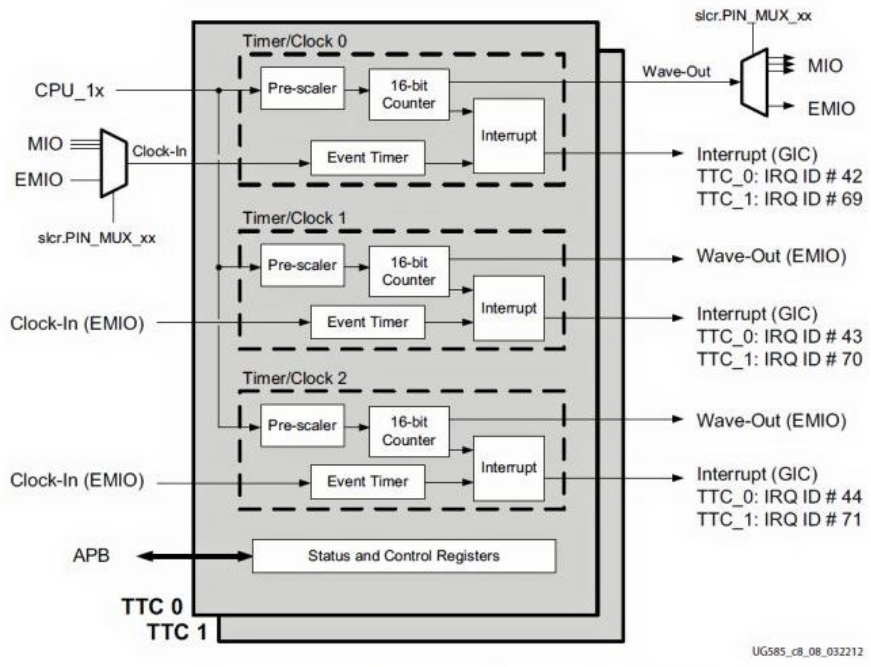

Figure 10: Triple Time Counter on Zedboard

```
{
  t = desc11[k] – desc22[k];
  dist += t*t;
}

dist_tot = sqrt(dist);
```

And the flowchart for calculating distance is as shown in Fig.-11: The other loop checks the distances as they are calculated and finds the minimum distance for kNN matching. dist1 is the previous minimum distance found and dist0 is the current one. If current distance is less than 0.8 times of old distance, then it is considered as a match case between the key points as mentioned before in previous sections. The distance along with the index at which it was found are stored. This code section is used for matching the keypoints later on the basis of whether the distances are within the threshold or not.

```
loop2: for(int i=0; i<number_keypoints1; i++){
  if(nidx[0] > 0){
       if(dist0 < (0.8f*dist1)) {
         matches_kaze.push_back(key_points11[i].pt);
         matches_kaze.push_back(key_points22[nidx[0]].pt);
   }
  }
}
```

Finding the minimum distance and its location is explained precisely in this flowchart shown in Fig.-12: We decided to implement the first loop in hardware as it is more time consuming and needs

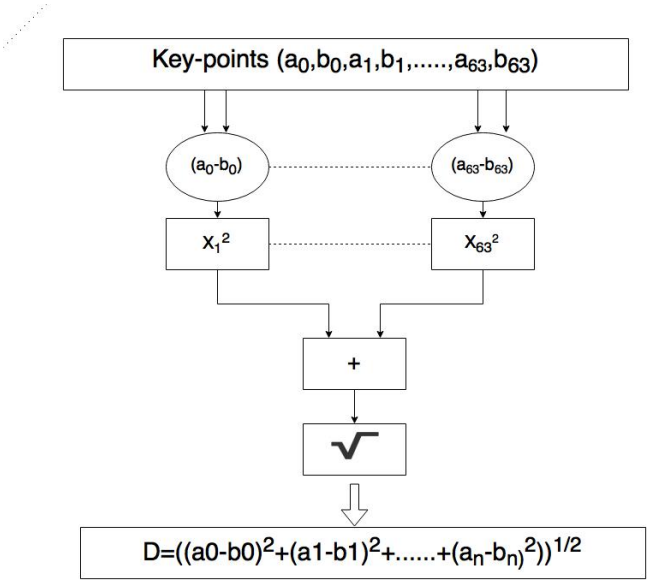

Figure 11: Flowchart for distance calculation between two keypoints

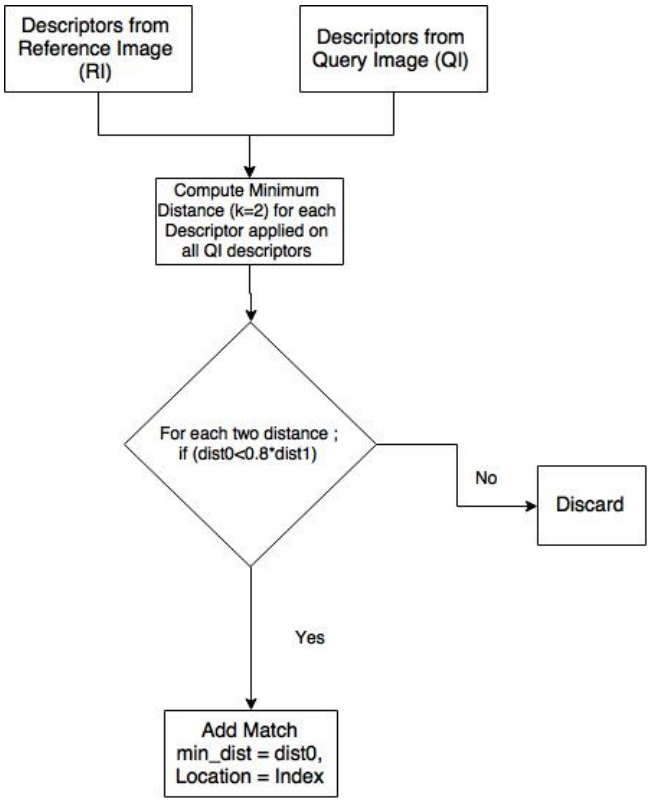

Figure 12: Flowchart for distance calculation between two keypoints

computation which can be effectively done using the ALU units of Zynq.

5.3 Conversion from C to VHDL code

The loop (loop1) computing distances needed to be converted into VHDL so as to be incorporated on the board. We used Vivado High Level Synthesis (HLS) to transfer code written in C into a VHDL code. To better utilize the parallelism of the loop, we used a directive called 'Loop Unrolling' provided by HLS. This is required as even though HLS tries to minimize latency on its own by pipelining functions, it does not schedule loops to operate in parallel by default.
The UNROLL directive tells HLS to try to execute the individual iterations of the loop in parallel. This is obviously very fast, but can cost a lot more hardware depending on the level of unrolling.
Another directive we used was 'Array Partition'. Partitioning breaks an array into smaller elements and is done to improve overall throughput.

- If the factor is not an integer multiple, the final array has fewer elements
- Arrays can be split along any dimension
  - If none is specified dimension zero is assumed
  - Dimension zero means all dimensions
- All partitions inherit the same resource target
  - That is, whatever RAM is specified as the resource target
  - Except of course "complete" [1]

The C to VHDL conversion resulted in four modules required for computing square root, multiplication, buffer in and out and the main module calling these modules. The flow chart shown in Fig.-13 describes the modules and the frequency of their usage to transform the loop. Fig.-13 also depicts the massive number of computations taken up by the loop for calculating distance and justifies that this section of matching algorithm indeed is a time consuming part. Comparing the number of times each module is called with the C code of loop1, we can see that dist += t*t requires multiplication module to be called 64 times (number of keypoint values in a row of descriptor array) whereas requires the Square root module to be called once as dist_tot = sqrt(dist) is done only once outside the loop to get the distance between two rows of query and train descriptors.

Fig.-14 shows the clock cycles taken without directives where as Fig.-15 shows the clock cycles taken with directives as a result of parallelization of distance calculation operation. Fig.-16 shows the resources consumed in case of without directives where as Fig.-17 shows the resources consumed with directives as a result of parallelization of distance calculation operation.

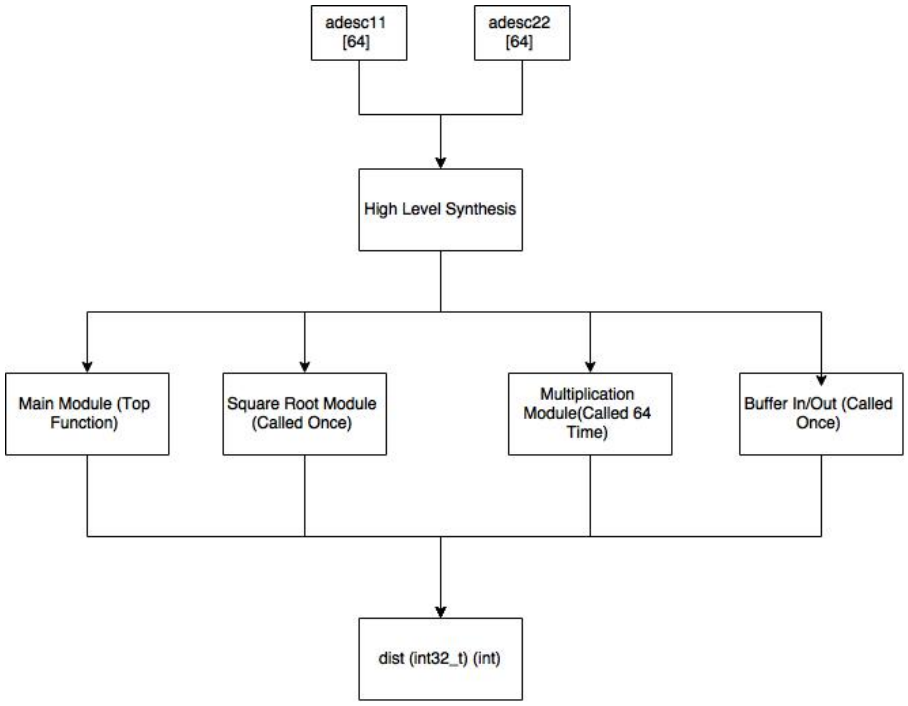

Figure 13: Hardware architecture of kNN accelearator

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.62	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
551	551	552	552	none

Figure 14: Performance without directive

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.62	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
46	46	47	47	none

Figure 15: Performance with directive

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	796
FIFO	-	-	-	-
Instance	-	4	2244	2825
Memory	-	-	-	-
Multiplexer	-	-	-	129
Register	-	-	286	-
Total	0	4	2530	3750
Available	280	220	106400	53200
Utilization (%)	0	1	2	7

Figure 16: Resource utilization without directive

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	4109
FIFO	-	-	-	-
Instance	-	256	2244	2825
Memory	-	-	-	-
Multiplexer	-	-	-	90
Register	-	-	1776	-
Total	0	256	4020	7024
Available	280	220	106400	53200
Utilization (%)	0	116	3	13

Figure 17: Resource utilization with directive

5.4 Creation of custom IP core

Using the VHDL code of distance calculation obtained using HLS with all modules (Main, Square root, Multiplication, Buffer In/out), we next created the IP core to be integrated with the Zynq board. This core performs distance calculation for a given pair of key point descriptors, one from TI ans the other from QI. This core is called repeatedly to achieve matching between the TI and QI. To handle this, we referenced the core inside a combination of two successive loops with the outer loop running for NK1 and inner loop for NK2. This ensures that each row of query descriptor is matched with every row of train descriptor. The subsequent part with finding the minimum distance and location was left to C code (loop2) as comparison doesn't take as much resources as multiplication and square root.

5.5 Steps for IP core creation

- With the base Vivado project opened, choose create and package IP. We use AXI Lite interface
- We then select 'edit IP' and add source VHDL codes generated from HLS
- We then instantiate the top VHDL file only (Main module)
- Portmap
  - Input in_std_logic or in std_logic_vector - slv_reg (n)
  - Output std_logic or std_logic_vector - user defined signals (Not slave registers)
- We also included the .tcl files as buffer in/out and square root modules were created as IP cores themselves
- Merge all the file groups
- Run synthesis
- Repackage IP

Our custom IP core is depicted as shown in the Fig.-18: And the overall Zedboard along with all components and custom IP is as shown in Fig.-19:

6 Conclusion

It can be seen that we have achieved the objective of parallelizing the kNN matching algorithm and demonstrated the effects on a conventional GPU and a FPGA platforms. Our future efforts would be include the feature into a stream of video such that the image recognition can take place between a given train image and the frames of the video. It would be interesting to see the performance of such

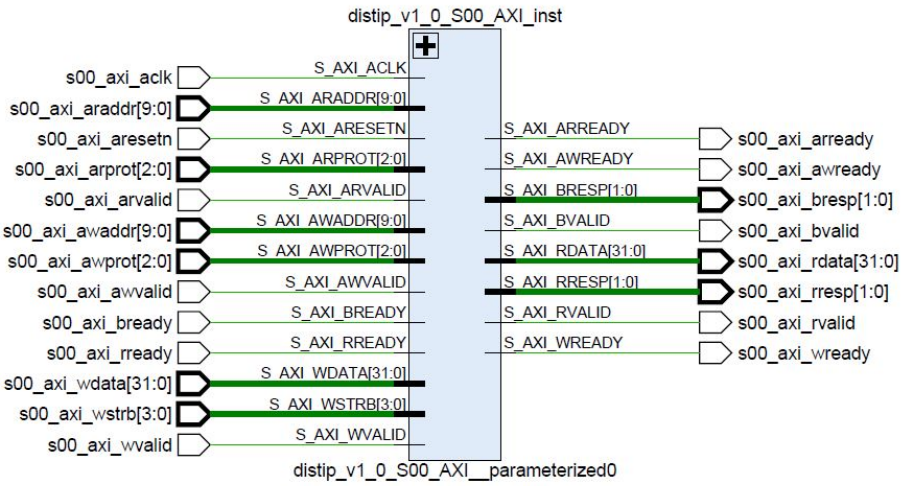

Figure 18: Custom IP core

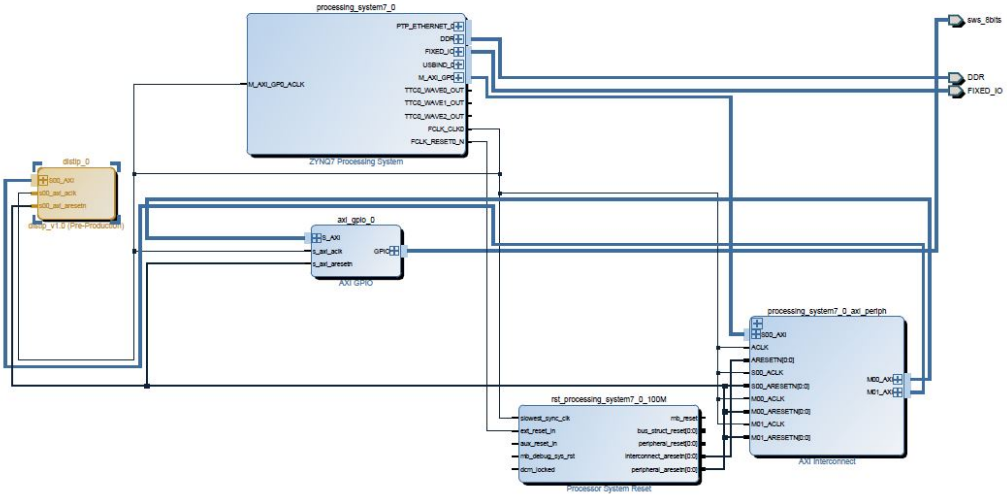

Figure 19: Custom IP core integrated into the Zedboard

a system. We have been convinced that hardware platforms such as GPU and FPGA would play a crucial role in the future in parallel computing where it is required to run independent threads.

References

[1] Intel Thread Building Blocks, https://www.threadingbuildingblocks.org/

[2] OpenCV home page, www.opencv.org

[3] http://www.em.avnet.com/en-us/design/drc/Pages/Zedboard.aspx