NCC Readme

Kimchyan Gan
Mao Zeng
Lucian Codrescu

1 Introduction

Normalized Cross Correlation (NCC) is one of fundamental operation in computer vision to find the coordinate or position of a given pattern in a region of an image. The region may include the whole image. The problem is also known as template matching, which matches the template (pattern) in a region or window of image. NCC is superior than other approaches like zero-mean correlation, sum square difference (SSD), sum of absolute differences (SAD) due to its insensitivity to the intensity of the pixel. Intensity differences can be created by different sensors or under different illumination intensities. One of application of template matching is in feature detection and tracking. For feature tracking, initial estimate of the feature is known in previous frame. The search of current frame starts at the estimated position of previous frame and its surrounding pixel.

The drawback of NCC is its computation complexity. Many papers, ranging from well-known Fourier-transform to pre-calculation of the sum and sum square tables, have been published to reduce its computation complexity such that real-time application is feasible. The complexity includes number of multiplications, computation of sum and sum square of an image, square root, etc. This memo is to discuss how to deal with these complexities and illustrates an efficient NCC implementation using the Hexagon Vector Extension (HVX).

Section 2 defines NCC and explains the term used in this memo. Section 3, 0, 0, and 6 illustrate each parts of NCC implemented on HVX of 64-byte vector length. Section 0

illustrates the NCC implementation on HVX of 128-byte vector length. Section 8 concludes with performance results.

2 Normalized Cross Correlation (NCC)

NCC finds the best match between a template (t) of $N_x * N_y$ size and a patch image ($f(x,y)$) within the given search windows of $M_x * M_y$. The NCC is defined as followed.

$$\gamma(u,v) = \frac{\sum_{x,y}[f(x,y) - \overline{f_{u,v}}][t(x-u,y-v) - \bar{t}]}{\{\sum_{x,y}[f(x,y) - \overline{f_{u,v}}]\sum_{x,y}[t(x-u,y-v) - \bar{t}]\}^{0.5}}$$

where $f(x,y)$is a value of image pixel at a given point (x,y), $t(x-u,y-v)$is the template of size $N_x * N_y$ shifted to (u,v), $\overline{f_{u,v}}$ is the mean value of $f(x,y)$ within the area of the template t, and $\bar{t}$ is the mean value of the template $t(x-u,y-v)$, $\sum_{x,y}[f(x,y) - \overline{f_{u,v}}]$ denotes the variance of $f(x,y)$, and $\sum_{x,y}[t(x-u,y-v) - \bar{t}]$ denotes the variance of the template, t.

The above equation is simplified, NCC can be defined as followed.

$$\gamma(u,v) = \frac{\sum_{x,y} f(x,y)t(x-u,y-v) - (N_x * N_y)\overline{f_{u,v}}\bar{t}}{\sigma_f \sigma_t}$$

where σ_f denotes the standard deviation of $f(x,y)$ within the area of the template t , σ_t denotes the standard deviation of the template.

The best match point, (u_b, v_b), can be found when the NCC function, $\gamma(u,v)$, attains its maximum value.

$$(u_b, v_b) = \arg\max_{u,v} \gamma(u,v)$$

.

2.1 NCC on HVX

NCC implemented on HVX is one of the functions from Qualcomm fast CV (Computer Vision) library. The template size, $(N_x * N_y)$, is a square of 8x8. The size of region of interest (ROI) in an image, $M_x * M_y$, is 18x18. Thus, 8x8 template matching in a region of 18x18 results in 11x11 search points. The higher level diagram of NCC is shown in Figure 1.

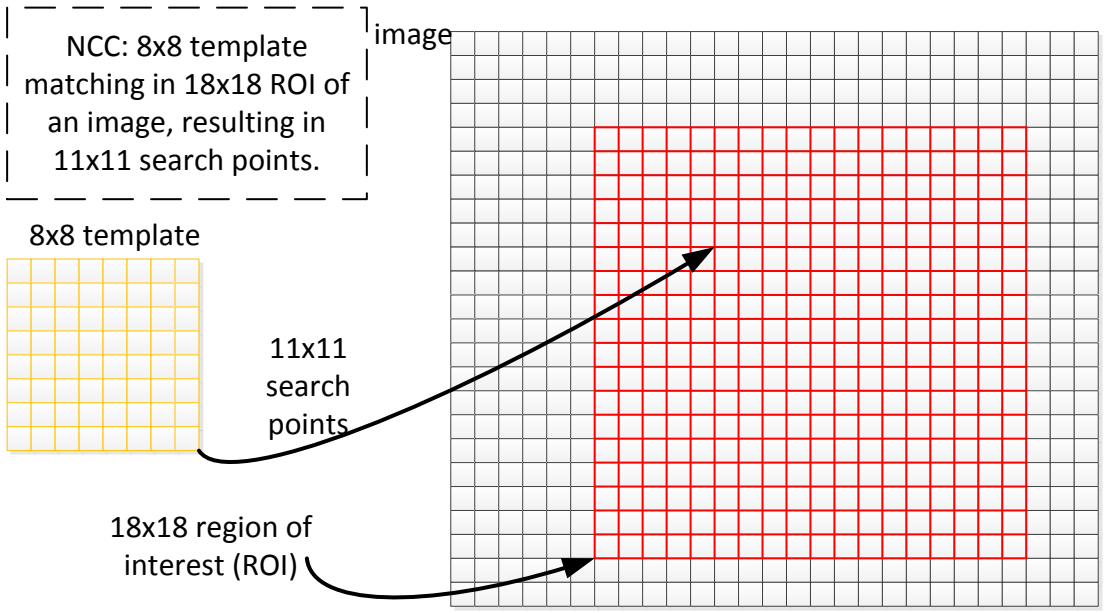

Figure 1: normalized cross-correlation (NCC) on HVX

When 11x11 searches are conducted, the template is constant. $\bar{t}$ and σ_t are constant throughout the search. These parameters can be computed once up front and use throughout the 121 (11x11) searches.

Implementation of NCC on HVX takes following steps.

1. Compute the sum $(64 * \bar{t})$ and sum of square of the template.
2. Compute 11x11 cross correlation by shifting the template to the left and/or to the bottom of the image. Compute the variance of the image under the template.
3. Find the coordinate or position, (u_b, v_b), of the best NCC. Best NCC corresponds to the maximum value of $\gamma(u, v)$.
4. Compute the NCC value, $128 * \gamma(u_b, v_b)$, at the best position in floating point.

At the end, NCC returns (u_b, v_b) and $128 * \gamma(u_b, v_b)$ to the caller.

HVX can be programmed with vector length of 64B or 128B. Most of the times, the same code can be utilized in either of modes, just by changing the vector length definition. In cross correlation and variance computation, the code is different for 64B and 128B. For all other steps, code can be shared between 64B and 128B.

3 Template Sum and Sum Square Computation

Template sum and sum square are computed upfront since they are constant through different shift in cross-correlation.

$$sumt = \sum_{x=u}^{u+7}\sum_{y=v}^{v+7} t(x-u, y-v)$$

$$sumt2 = \sum_{x=u}^{u+7}\sum_{y=v}^{v+7} t^2(x-u, y-v)$$

When mapping the above operation to HVX, it becomes an unaligned load followed by multiply and vector reduction. Vector reduction involves permute unit (VALIGN instruction) to allow data in different lanes to merge. Merging is expensive due to 1 cycles latency in permute instructions. It also involves moving the data from the vector back to the scalar, which takes tens of cycles due to post-commit architecture on HVX.

If this part is implemented on Q6 scalar core (which contains 8-byte vector), it can be accomplished in 11 cycles. No dealing with instruction latency and data movement are needed. On Q6 scalar, VRADDUB instruction is suitable for sum computation and VRMPYBU is used to compute sum of square.

4 Cross-Correlation and Variance Computation

The first step in this process is to spread the $f(x,y)$ to a format friendly for 32-bit accumulator on HVX. The input is 18 8-bit pixels for a row, data after 18 bytes in the register can be ignored. Through permute instruction of VDELTA, the 18 bytes input is spread into two vectors, and each is with groups of 4, as shown in Figure 2. For two vectors output generated from permute instruction, only the first 11 32-bit are used, leaving the following 5 32-bit unused, because the search point is 11 in width.

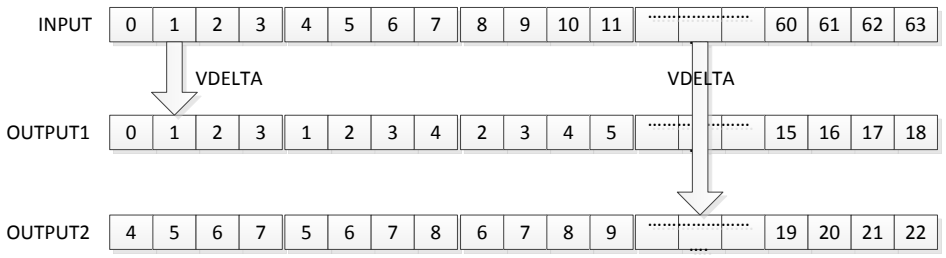

Figure 2: two permutes using VDELTA instruction

Once the input is permuted into friendly format, $sumf(u,v)$ and $sumf2(u,v)$ can be computed.

$$sumf(u,v) = \sum_{x=u}^{u+7}\sum_{y=v}^{v+7} f(x,y)$$

$$sumf2(u,v) = \sum_{x=u}^{u+7}\sum_{y=v}^{v+7} f^2(x,y)$$

When using HVX, the registers are big enough to hold all the entries of 8x8. Only for the initial vertical shift ($v=0$), all the terms of $sumf(u,v)$ or $sumf(u,v)$ need to be added (8 rows and each row has 8 entries located in two vector registers). For the rest of shift ($v=1:10$), windowing running sum approach can be used. Basically, the oldest row entry is subtracted out and the newest row entry is added. In this way, a row of computation only involves two vector subtractions (oldest row with 0-3 and 4-7) and two vector additions (newest row with 0-3 and 4-7). So, $sumf(u,v)$ or $sumf2(u,v)$ takes about 2 packets to compute using running sum (instead of 8 packets).

Next, the zero-mean cross correlation can be computed and its formula is show as below.

$$noms(u,v) = \sum_{x,y} f(x,y)t(x-u,y-v) - \big((sumt * sumf) \gg 6\big)$$

If 8 rows of inputs are kept in the registers, the bottleneck is in the permute unit. For every single row of input, it requires two permutes, thus resulting in 16 packets for 8 row of processing. Instead, all the permuted input are kept in registers, and it requires 16 registers instead of 8 registers. Also, the delay lines require 14 shifts (instead of 7).
The cross-correlation uses 8-bit multiply and HVX can issue up to two 8-bit multiply instructions in a packet. To pack the computations into two multiply instructions per packet, split computation technique is used where one accumulates the indices from 0 to 3 and another from 4 to 7. This part is one of the intensive part of NCC and the code is shown in Figure 3.

```
{ sCross0.uw = VRMPY(sLine0_03.ub,t0_03.ub)     //[1]
  sCross1.uw = VRMPY(sLine0_47.ub,t0_47.ub)     //[1]
}
{ sCross0.uw += VRMPY(sLine1_03.ub,t1_03.ub)    //[1]
  sCross1.uw += VRMPY(sLine1_47.ub,t1_47.ub)    //[1]
}
{ sCross0.uw += VRMPY(sLine2_03.ub,t2_03.ub)    //[1]
  sCross1.uw += VRMPY(sLine2_47.ub,t2_47.ub)    //[1]
}
{ sCross0.uw += VRMPY(sLine3_03.ub,t3_03.ub)    //[1]
  sCross1.uw += VRMPY(sLine3_47.ub,t3_47.ub)    //[1]
}
{ sCross0.uw += VRMPY(sLine4_03.ub,t4_03.ub)    //[1]
  sCross1.uw += VRMPY(sLine4_47.ub,t4_47.ub)    //[1]
}
{ sCross0.uw += VRMPY(sLine5_03.ub,t5_03.ub)    //[1]
  sCross1.uw += VRMPY(sLine5_47.ub,t5_47.ub)    //[1]
}
{ sCross0.uw += VRMPY(sLine6_03.ub,t6_03.ub)    //[1]
  sCross1.uw += VRMPY(sLine6_47.ub,t6_47.ub)    //[1]
}
{ sCross0.uw += VRMPY(sLine7_03.ub,t7_03.ub)    //[1]
  sCross1.uw += VRMPY(sLine7_47.ub,t7_47.ub)    //[1]
}
{ sCross0.w = VADD(sCross0.w,sCross1.w)         //[1] combine results from split
}                                               //    computation
```

Figure 3. 8x8 Cross-correlation on HVX which takes 8 packets

At last, the variance of image under the template is computed, as followed.

$$denoms(u,v) = sumf2(u,v) - ((sumf(u,v) * sumf(u,v)) \gg 6)$$

The $noms(u,v)$ and $denoms(u,v)$ corresponds to cross-correlation and variance of image, respectively. Both are the output to the temporary memory for next stage to process.

Since NCC can start in arbitrary point within an image, no assumption for the alignment of image data can be made. HVX architecture provides unaligned load instruction (VMEMU) which comes in very handy in this routine. This instruction takes up two slots and only additional 2 (instead of 3) instructions can be packed together with unaligned load instruction in a packet. This instruction loads VLEN bytes, where VLEN can be 64

or 128, even though only 18 pixels will be used. If the load happens at the end of image, memory access may be over buffer boundary, triggering access violation. So, it is required to allocate additional VLEN bytes for the image used in NCC function.

For each vertical shift processing, only 11 32-bit $noms(u,v)$ or $denoms(u,v)$ are output. In other words, the first 44 bytes (11 32-bit) of vector contains meaningful data. To compact all the output in a contiguous memory, unaligned store instruction is used. The output contains 121 (11*11) entries. The buffer length is allocated to the ceiling of multiple of VLEN, which is 512 bytes for 64-B HVX. So, the last few entries (entries 122 to 128) is useless but they need to be zero out for efficient processing in next stage.

Last but not least, the buffer allocated by caller to $noms(u,v)$ or $denoms(u,v)$ is 512 bytes for 64-B HVX but 640 bytes (512+VLEN) for 128-B HVX. Why do we additional VLEN bytes for 128B HVX? The answer is depicted in Figure 4. The unaligned save instruction is used for compacting all the search results in different vertical shifts. In 128 B mode, the vector can contains up to 32 output per vector, with each output being 4 bytes. When processing the last vertical shift, NCC generates 11 output and 21 data following are not required. When writing the last entries, the code may write over the 512-byte boundary. Over allocating the buffer by VLEN is one of solution.

Another solution for this issue is to emulate the byte enable store. On HVX, there no byte-enable store with unaligned support. The output of last vertical shift can be muxed with second to the last output before storing in memory.

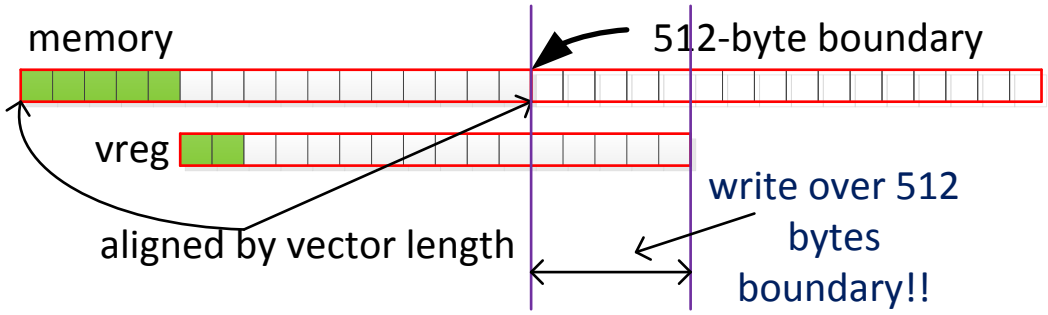

Figure 4: Last unaligned store writes over 512-byte buffer boundary.

5 Finding the Best Match

When finding the best match, 121 $\gamma(u,v)$ will be computed and the index corresponding to maximum value of $\gamma(u,v)$ is returned. The first challenging part is to compute square root of variance to obtain the standard deviation. To avoid approximation or lengthy computation of square root, the formula is modified by squaring the criteria. The modified criteria becomes as followed.

$$\gamma'(u,v) = \frac{N}{D} = \frac{noms(u,v) * noms(u,v)}{denoms(u,v)}$$

No square root is needed. It, however, leads to higher precision of 64 bit requirement when $noms(u,v) * noms(u,v)$ is computed. There is no native HVX instruction to construct full 32x32 multiplication output. However, there is instruction for fractional and integer 32x32 multiplication. In this case, factional multiplication and shift instructions are used to constructed higher 32-bit output and integer multiplication is used to constructed lower 32-bit output. The $noms(u,v)$ in NCC may take up to 22 bits. The square of it will take at most 44 bits. The range of it will ensure the fractional multiplication will never overflow when creating the upper 32 bits of output.

```
{ sH32nn.w = VMPYE(sN.w,sN.uh)                  //[1] upper 32 bits of 32x32
}
{ sH32nn.w += VMPYO(sN.w,sN.h):<<1:sat:shift    //[1] upper 32 bits of 32x32
}
{ sH32nn.w = VAVG(sH32nn.w,sZero.w)             //[1] upper 32 bits of 32x32
  sL32nn.w = VMPYIEO(sN.h,sN.h)                 //[1] lower 32 bits of 32x32
}
{ sL32nn.w += VMPYIE(sN.w,sN.uh)                //[1] lower 32 bits of 32x32
}
```

Figure 5: Compute square of $noms(u,v)$, which is 44-bit output

After creating the 64-bit numerator and 32-bit denominator, the next step is division which is notoriously slow in embedded processing. How can the division be avoided? Since the operations compares which side has larger value, the denominator of one side can be move to another side through multiplication.

$$\frac{N_i}{D_i} > \frac{N_{max}}{D_{max}} \Rightarrow N_i * D_{max} > N_{max} * D_i$$

It leads to 64x32 multiplication. The N is actually 44 bits and D needs 22 bits, thus, total output from the multiplication is 66 bits. So, only 80-bit of the output is computed (ceiling to closer multiple of 16). The upper 16 bits (bit 64 to 79) and lower 32 bits (bit 0 to 31) are easy to compute. The tricky part is the middle 32 bit (bit 32 to 64), since there is no U32*U32 supports. Instead, I32*I32 instruction is used, and the input signed bit is

masked out and add back later on. Also, the middle 32-bit output consists sum of two parts, as shown in Figure 6. In short, to compute 80-bit $N * D$, a total of two integer 32x32, one fractional 32x32, one fractional 32x16, and some patched instructions are needed. The actual instructions used to compute the output is shown in Figure 7.

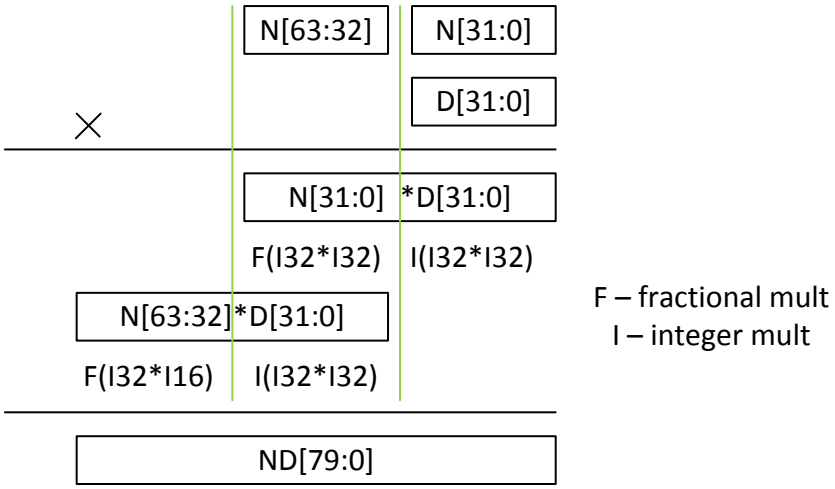

Figure 6: multiplication of *N*[43:0]x*D*[21:0], creating 80-bit output

```
////////////////////////////////////////////////
// compute the upper 16 bits of output - Out[80:64]
////////////////////////////////////////////////
sHnnpRight.w = VMPYE(sD.w,sBestH32nn.uh)      //[1]
sHnnpRight.uw = VLSR(sHnnpRight.uw,R7)        //[1] >> 16

////////////////////////////////////////////////
// compute the middle 32 bits of output - Out[63:32]
////////////////////////////////////////////////
sBestL32nnM = VAND(sBestL32nn,sMask)          //[1] mask out signed bit
Q3 = VCMP.EQ(sBestL32nn.w,sBestL32nnM.w)      //[1] is sign bit contain value?
sMnnpRight.w = VMPYE(sD.w,sBestL32nn.uh)      //[1]
sMnnpRight.w += VMPYO(sD.w,sBestL32nnM.h):<<1:sat:shift//[1]
IF (!Q3) sMnnpRight.w += sD.w                 //[1] patch I32*I32 to U32*U32
sMnnpRight.w = VAVG(sMnnpRight.w,sZero.w)     //[1]
sMnnpRight.w += VMPYIE(sD.w,sBestH32nn.uh)    //[1]

////////////////////////////////////////////////
// compute the lower 32 bits of output - Out[31:0]
////////////////////////////////////////////////
sLnnpRight.w = VMPYIEO(sD.h,sBestL32nn.h)     //[1]
sLnnpRight.w += VMPYIE(sD.w,sBestL32nn.uh)    //[1]
```

Figure 7: Compute 80-bit output from 64x32 multiplication

The output $N * D$ is stored in 3 vector registers, containing a total of 96 bits per element, even though the most significant 16 bit of upper register will always be 0. Now, 96-bit output will be compared with each other and the larger one will be kept. There is no native 96-bit comparator, and cascaded comparison is implemented to select the larger

output, as shown in Figure 8. The basic idea is to compare the upper 32 bits first. If the right operand is bigger, it will be selected. If they are the same, compare the middle 32 bits. Same logic can be applied when comparison of lower 32 bits is needed.

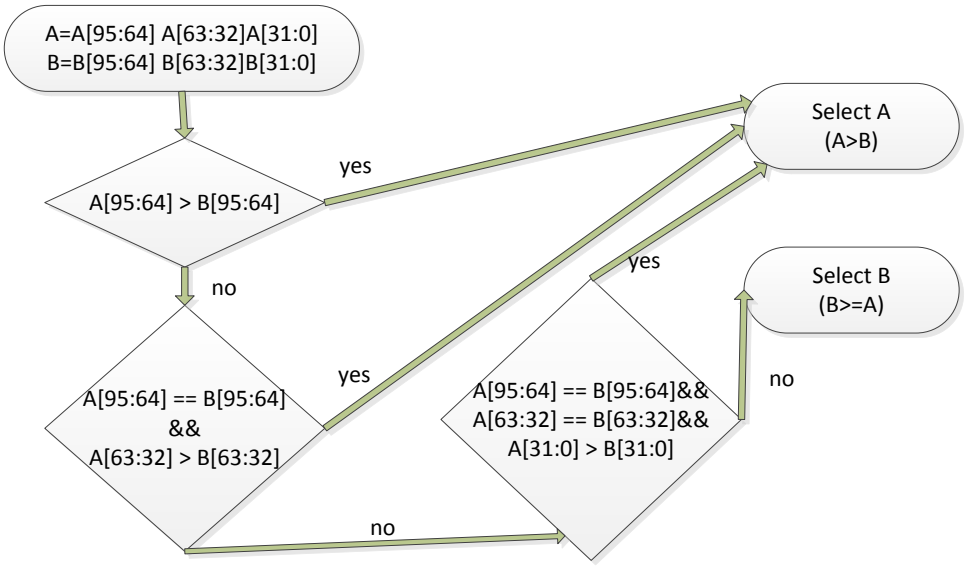

Figure 8: Cascaded comparison of two 96-bit values to select a larger one

First part of finding the best match involves consuming 121 inputs and picking the best NCC values per vectors. Second part involves in vector reduction which selects the best NCC values within a vectors.

6 Compute Best NCC Value

Once the best point is known, NCC computes $128 * \gamma(u_b, v_b)$ and returns the best NCC values. To generate the bit exact result as fast CV library, this part is computed in floating point. The main computations involve the floating-point math library square root and floating-point division.

7 NCC Using 128 Bytes of Vector Length

When extending NCC from 64B to 128B vector, all the functions can be reused, except cross-correlation and variance computation block. This section only discusses this block for 128B vector implementation.

Since the cross-correlation is 8x8 block processing, it is no easy to fully utilize all the multiply in computing the cross-correlation. When doing cross-correlation in 64B vector, 31% (5/16) of the multiply is essentially wasted and not used. If the same code is used for 128B, waste will increase to 65% ((5+16)/32). Essentially, more than half vector is not used.

One of idea to increase the 128-B usage is simply spreading the input into group of 8 (instead of group of 4), thus, computing a row of cross-correlation in one VRMPY instruction. However, this will not work since HVX VRMPY instruction does not take scalar register pair (Rss) which allows 8 bytes of template to be stored. Another idea is computing two adjacent vertical shift of the samples using one vector since both sample shares the same input template data. This approach does allow us to use additional elements in a 128-byte vector and the waste in a vector is the same as 64-B counterpart -- 31% (10/32). However, additional instructions are needed to mux two rows of data into a vector register. Since two row samples are computed, delay lines become 12 instead of 14. The vertical shift is 11 (odd number), so half of the vector length is still wasted in last vertical shift of computation.

The breakdowns of profiling information for 64B and 128B vector are shown in Table 1. Some of functions like floating point computation and "integrate64u8" do not scale at all from 64B to 128B since they run on Q6 scalar. "computecrossvar" only has 21.3% reduction mainly due to 8x8 processing block. Even though two output lines are processed at simultaneously in this function, the overhead and effort to mux two lines into vector registers hurt performance. The vertical shift is odd, thus, half of vector is used in last vertical shift. As for "searchbestncc", the first part scales well with vector, second part is vector reduction which yields no improvement from 64B to 128B.

Table 1. The profiling information of 64B and 128B vectors

NCC functions	T-cycles for 64B	T-cycles for 128B	reduction
integrate64u8	11.0	11.0	0.0%
Computecrossvar	207.0	163.0	21.3%
Searchbestncc	204.9	161.9	21.0%
Misc (scalar computation, float sqrt root, etc)	--	--	0.0%
End-to-end NCC	523.1	436.1	16.6%

8 Performance

Table 2 summarizes the performance of NCC measured on simulator.

Table 2. Performance of NCC on simulator

Pcycles/search	4T-64B	2T-64B	2T-128B
Ideal	276.5	540.6	453.7
Timing	328.9	595.4	521.5

It's obvious from the table that 4T-64B yields to the best performance. The p-cycle reduction from 2T-64B to 4T-64B is around 45%. If mode switching is allowed and no other workload occupies another two threads, 4T-64B is the best choice. In typical cases, two threads are occupied by the audio workloads, only two more threads are available, the best choice is 2T-128B. The reduction from 2T-64B to 2T-128B is only 16.6%.

In actual application, 2T-128B of NCC is most likely to be used and it consumes 453.7 packets per search. In the timing simulation, it is assumed the data is not in the L2 cache and need to be pre-fetched into L2 cache. In actual application, the data may already in L2 cache due to coarse search for the features in computer vision. The pre-fetch is issued in multi-threading C code. So, if no pre-fetch is required, there is no need to change in assembly code.

Last but not least, one of code that runs on Q6 scalar is math library square root function. If the precision can be relaxed, using native floating-point reciprocal square root approximation instruction (SFINVSQRTA) can further reduce 11 cycles per search. The precision relaxation does not affect the precision of best match position (u_b, v_b) but only the NCC value $(128 * \gamma(u_b, v_b))$. So, it is fine for most of the application.

Reference

1. Lewis, J.P., *Fast Normalized Cross-Correlation*.
2. Briechle, Kai and Hanebeck, D. U, *Template Matching using Fast Normalized Cross Correlation*.
3. http://en.wikipedia.org/wiki/Cross-correlation
4. http://docs.opencv.org/doc/tutorials/imgproc/histograms/template_matching/template_matching.html