PERFORMANCE ANALYSIS OF H.263 VIDEO ENCODER FOR VIRAM

by
Thinh PQ Nguyen

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science, Plan II
University of California at Berkeley

November 1999

Approved by ____________________________________
Professor Avideh Zakhor, Research Advisor

Date ____

Approved by ____________________________________
Professor Kathy Yelick, Second Reader

Date ____

PERFORMANCE ANALYSIS OF H.263 VIDEO ENCODER FOR VIRAM

Thinh PQ Nguyen

M.S. Report

Abstract

VIRAM (Vector Intelligent Random Access Memory) is a vector architecture processor with embedded memory, designed for portable multimedia processing devices. Its vector processing capability results in high performance multimedia processing, while embedded DRAM technology provides high memory bandwidth at low energy consumption. In this thesis, we evaluate and compare performance of VIRAM to other digital signal processors (DSPs) and conventional SIMD (Single Instruction Multiple Data) media extensions in the context of video coding. In particular, we will examine motion estimation (ME) and discrete cosine transform (DCT) which have been shown to dominate typical video encoders such as H.263. In doing so, we point out the advantages and disadvantages of certain VIRAM's designs with respect to video coding. In addition, we also show that VIRAM outperforms other architectures by 4.6x to 8.7x in computing motion estimation and by 1.2x to 5.9x in computing discrete cosine transform. With appropriate VIRAM's features, our simulation shows that VIRAM can achieve near real-time encoding of standard video QCIF sequences using exhaustive search for motion estimation.

TABLE of CONTENTS

1 Introduction

Traditionally, video processing is performed by high-end workstations or specialized hardware such as ASICS. Recent popularity of portable and hand-held devices such as digital cameras and wireless videophones has created a need for hardware architectures designed for mobile, portable video processing applications [13]. Conventional microprocessors are not well suited to video processing because they are optimized for traditional applications with complex control flow. On the other hand, the kernels of multimedia applications are often characterized by large amounts of data parallelism and high memory bandwidth [2]. For instance, standardized video codecs such as MPEG-4 and H.263 consist of motion estimation (ME) and discrete cosine transform (DCT), both requiring high memory bandwidth and involving large amounts of data parallelism. In this thesis, we analyze the speed performance of an H.263 encoder on VIRAM (Vector Intelligent Random Memory Access), the vector microprocessor being designed at U.C. Berkeley for multimedia applications on portable devices. By integrating vector processing with embedded DRAM technology, VIRAM eliminates off-chip memory accesses, and therefore achieves high memory bandwidth at low power consumption. In addition, VIRAM's vector architecture provides high parallel processing power that is suitable for multimedia applications.

The H.263 is the ITU (International Telecommunication Union) recommended standard for very low bit rate video compression. We choose to analyze the performance of H.263 video codec because it has lower computational and power requirements compared to other algorithms, and therefore is well suited for portable devices.

The remainder of the thesis is organized as follows. Section 2 gives a brief introduction to H.263 video encoder. Section 3 gives an overview of VIRAM architecture. In section 4 we characterize the time distribution of individual components of H.263 video encoder, and lead to the discussion of optimizing motion-estimation and performance results for VIRAM in section 5. Section 6 is devoted to optimized algorithms and speed performance results of the discrete cosine transform for VIRAM. In Section 7 we present the overall encoding speed of H.263 on VIRAM. Finally, we conclude in Section 8.

2 H.263 Overview

The H.263 encoder can be divided into five logical parts: motion estimation, discrete cosine transform, quantization, reconstruction, and entropy coding. Figure 1 shows a logical diagram of an H.263 encoder.

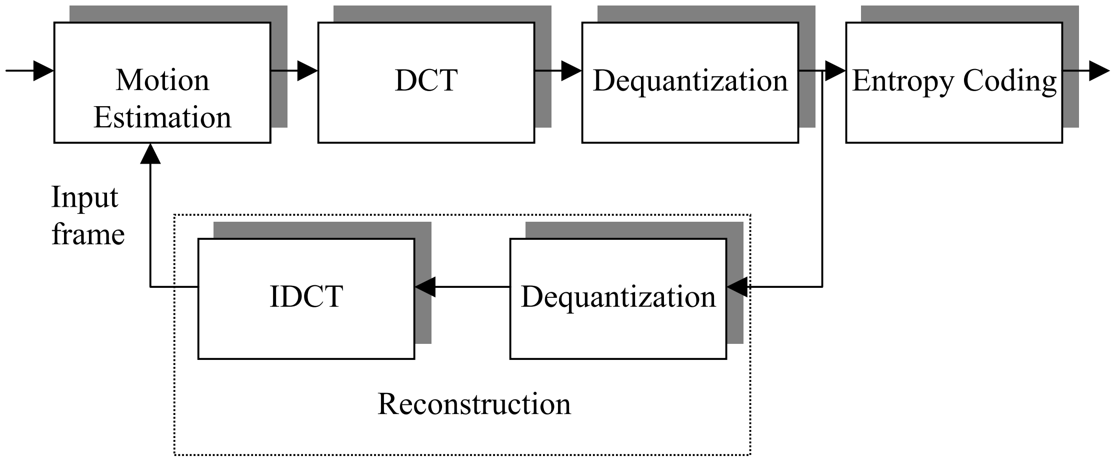


Figure 1. *Block diagram of H.263 encoder*

An input video sequence for an H.263 video encoder consists of frames. Each frame contains luminance and chrominance information. Because human visual system is less sensitive to chrominance, the chrominance information is subsampled by a factor of two on both horizontal and vertical directions. Each frame is also further divided into "macroblocks". A macroblock consists of one 16x16 luminance pixels and two corresponding 8x8 chrominance pixels. A frame can be I-frame (intra-frame), P-frame (predicted inter-frame), or PB-frame (bi-directional predicted frame). I-frame is encoded without motion estimation, while P-frame and PB frame are encoded using motion estimation to reduce temporal redundancy. In this thesis, we work with H.263 base-line encoder written by Telenor [16] that only considers I-frame and P-frame.

Motion estimation is used to reduce temporal redundancy in a video sequence. The hypothesis is that the current frame and the previous frame are similar, hence by coding their differences, the compression efficiency is higher than coding the actual pixel values of the current frame. To reconstruct the current frame, the differences are added to the pixel values of the previous frame. Therefore, the goal of motion estimation is to find the 16x16 region in previously reconstructed frame that is most similar to the macroblock in the current frame. This method minimizes the differences, hence achieves higher compression ratio. Generally, similarity is defined based on the sum of absolute differences (SAD) of luminance values between the macroblock of current frame (reference frame) and the 16x16 region in previously reconstructed frame. The lowest SAD value is the best match. Since all I-frames are reference frames, motion estimation applies only to the macroblocks of the P-frame (inter-macroblock). Motion estimation also uses only the 16x16 pixel luminance square of the macroblock, ignoring the chrominance information. The output of the motion estimation for each inter-macroblock are: the motion vector (MV) which indicates the relative location of the macroblock in the current frame with the best matched 16x16 region in the previously reconstructed frame, and their differences in pixel values, also called prediction errors.

After motion estimation, DCT is applied to the prediction errors of inter-macroblocks. Since motion estimation is not used on intra-macroblocks, DCT is directly applied to the luminance values of intra-macroblocks. DCT is used to transform pixel values in spatial domain to frequency domain for energy compression. Next, the outputs of DCT are quantized into discrete values, frequently referred to as reconstruction levels. The step size between two consecutive reconstruction levels is 2Q. In H.263, Q ranges from 1 to 31. Before sending the quantized DCT coefficients to the entropy encoder, reconstruction of the current frame is needed to provide the reference frame for motion estimation of the next frame.

To reconstruct the current frame, each macroblock of the current frame is reconstructed. The pixel values of the intra-macroblock or the prediction errors of the inter-macroblock is obtained by dequantizing the quantized DCT coefficients then applying the IDCT (inverse DCT) to the dequantized DCT coefficients. If the macroblock is an intra-macroblock, the reconstructed macroblock contains the direct outputs of the IDCT (pixel values). If the macroblock is an inter-block, the outputs of IDCT (prediction errors) are added to the pixel values of the macroblock in the previous frame (used as reference frame) to obtain the reconstructed-macroblock.

Finally, the quantized DCT coefficients are sent to the entropy encoder such as Huffman encoder. The output is a sequence of data bits and information with appropriate format for the decoder.

3 Overview of VIRAM Architecture

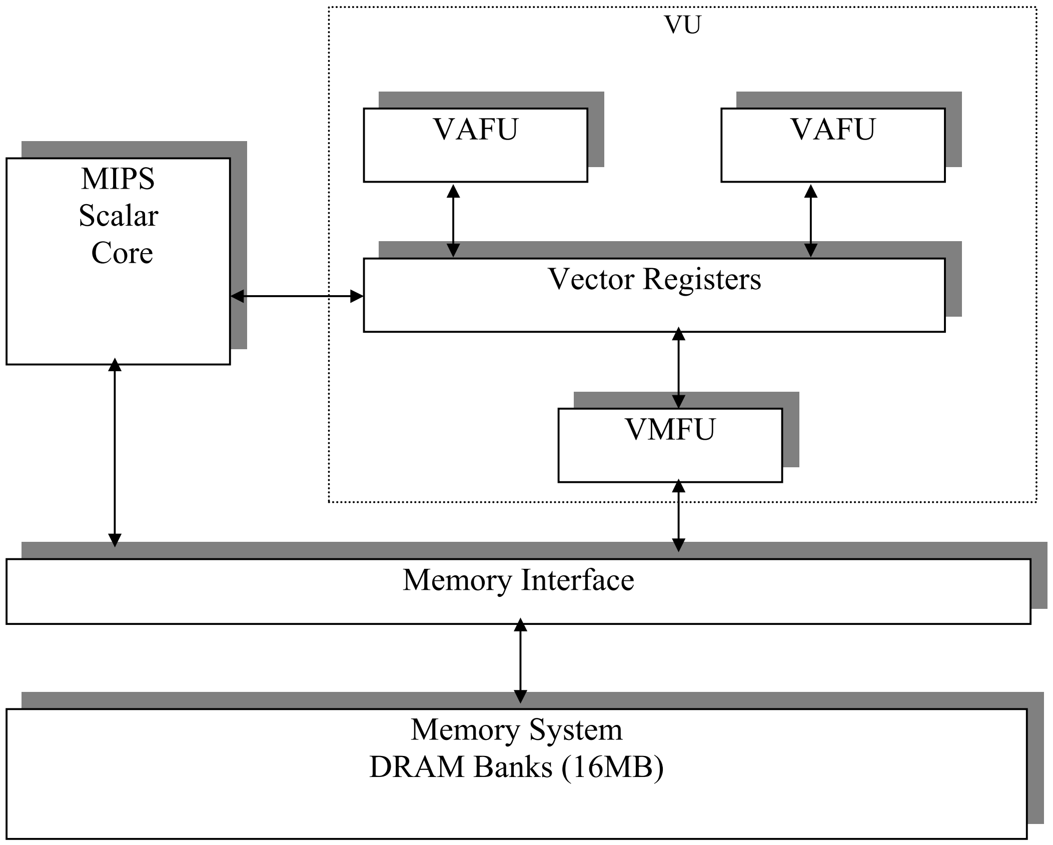


Figure 2. *Simplified block diagram of VIRAM architecture*

VIRAM is a vector microprocessor with on-chip main memory, designed for media processing. In his master's thesis [13], Kozyrakis explains in detail the designs of VIRAM. In this thesis, we present a brief introduction to VIRAM. Figure 2 shows the block diagram of VIRAM architecture. It contains a MIPS core scalar unit and a loosely coupled vector unit (VU). It is being designed using 0.18 µm embedded DRAM technology with target clock rate of 200MHz and 1.2V power supply. Because the processor and main memory are placed on the same chip, VIRAM can potentially increase memory bandwidth by 100 times as compared to conventional microprocessor systems [15]. The target power consumption for the vector unit and memory is 2 watts, which is suitable for portable devices [13,5].

VIRAM has two vector arithmetic functional units (VAFU) and one vector memory functional unit (VMFU). Both VAFU and VMFU have four 64-bit vector data paths that can be used to perform eight 32-bit or sixteen 16-bit operations per cycle. Both VAFUs support integer, fixed point, logical operations but only one supports floating-point operations. VIRAM peak performance is 6.4 GOPS for 16-bit data type, 3.2 GOPS for 32-bit data type, and 1.6 GOPS for 64-bit data type. Besides vector arithmetic and logical instructions, VIRAM also supports a wide range of scalar–vector instructions which have a scalar and a vector as operands. Most instructions are fully pipelined. The VMFU can load and store up to 256 bits per cycle. There are three types of vector memory accesses: *unit stride* which accesses contiguous memory locations, *strided* which accesses memory locations by a fixed offset, and *indexed* which accesses memory locations pointed by elements in a vector register. VIRAM's register file contains 32 vector registers and 32 scalar registers. Each vector register is 2048 bits long. VIRAM's memory system has 16 Mbytes of DRAM organized into eight banks. Finally, there is an I/O interface with four 100MB/s parallel lines. Since VIRAM is not yet available, the performance results in this thesis are based on a near cycle-accurate simulator that was developed at U.C. Berkeley [6].

4 H.263 Performance Characterization

To characterize the performance of the encoder, we use the H.263 version 2 written by Telenor [15], a popular public-domain implementation of H.263. Our test environment is a SGI machine running at 180MHz. We first optimize the H.263 encoder by changing many common functions into macros. We also replace the slow IDCT with the fast IDCT provided with the Telenor source code. During the measurements, we minimize the computational load of the machine by not running any other programs. Our tests include four QCIF (176x144) standard H.263 test sequences: Akyio, Mom, Hall, and Foreman. Each sequence contains 300 frames. We use quantization level Q = 10, and target frame rate of 10fps for all the test sequences. To measure the time spent on memory and arithmetic operations only, each test sequence is run from 3 to 5 times until the total time converges to a stable value. This method avoids the influence of disk accesses since all the data are already cached in the memory from the previous runs.

Sequence	Motion estimation	DCT and IDCT	Other	Total
Akiyo (12.95 kbit/s)	18765 (80.9%)	2306 (9.9%)	2130 (9.1%)	23201
Mom (16.25 kbit/s)	22446 (82.3%)	2508 (9.2%)	2310 (8.4%)	27264
Hall (20.47 kbit/s)	17745 (79.5%)	2282 (10.2%)	2300 (10.3%)	22327
Foreman (65.52 kbit/s)	27367 (82.8%)	2967 (9.0%)	2706 (8.2%)	33040

Table 1. *Distribution of time spent on individual components of H.263*

As shown in Table 1, motion estimation and DCT dominate around 81% and 9.5% of the total encoding time, respectively. Since they are also highly vectorizable, we will optimize the H.263 encoder for VIRAM, based entirely on motion estimation and DCT.

5 Motion Estimation

Motion estimation is used to exploit the inherent temporal redundancy of video [7]. In a typical motion estimation process, each frame of the video is divided into 16x16 macroblocks as described in Section 2. Given a macroblock in the current frame, the goal is to find the 16x16 region from the previously reconstructed frame that is most similar to it as shown in Figure 3. In general, the similarity is defined based on the minimum value of Sum of Absolute Value (SAD) between the luminance pixel values of the two 16x16 blocks:

$$SAD(x,y,k,l) = \sum_{j=0}^{15}\sum_{i=0}^{15} | F_1(i+x, j+y) - F_0(k+i, l+j) |$$

where (x,y) and (k,l) are the lower left-corner positions of the current macroblock and the 16x16 region in the previously reconstructed frame, respectively, and F_1 and F_0 are the pixel luminance values of current frame and previously reconstructed frame.

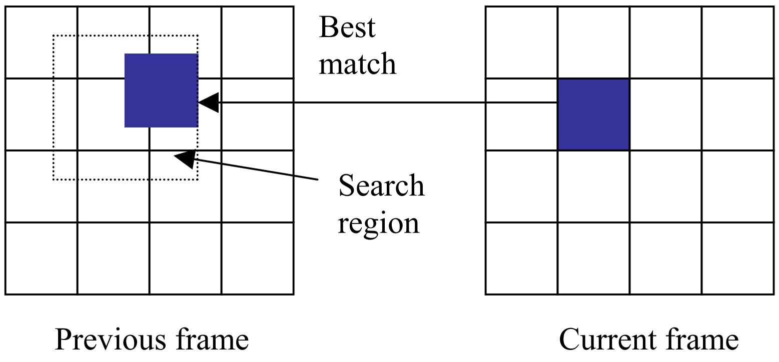


Figure 3. *Each macroblock (16x16 pixels) in the current frame is compared with 16x16 region in the previous frame to find the best match.*

Motion estimation is quite slow using exhaustive search where the SAD of all the possible 16x16 squares in the search region are computed to find the best match. Therefore, other motion estimation techniques have been proposed such as three-step search [11] and two-dimensional logarithmic search [10] to reduce the time complexity. These techniques compute SAD only at selected locations rather than all the possible 16x16 squares in the search region. As a result, they are faster to compute at the expense of lower picture quality. SAD, however, is still the dominant operation during motion estimation. Therefore, most DSPs and multimedia extensions vectorize the SAD function. In the next two subsections, we will describe two vectorized algorithms for computing motion estimation using exhaustive search. The first algorithm is based on vectorizing SAD within the macroblock, and in the second algorithm, data across the search region are vectorized. We then show the advantages and disadvantages of the two algorithms.

5.1 Motion Estimation Algorithm by Vectorizing SAD within a Macroblock

5.1.1 Description

Vectorizing SAD function requires the *reduction* operation, which takes a vector and reduces to a scalar value by summing all the elements of the vector, as shown in Figure 3.

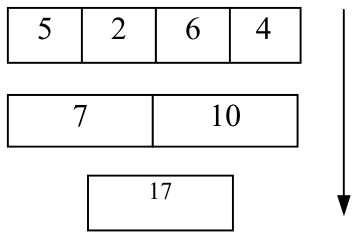


Figure 3. *Reduction instruction*

Generally, a *reduction* operation is expensive as it breaks the flow of the vector pipeline. For motion estimation using exhaustive search, the cost is high because the *reduction* operation is inside the inner loop as shown in the below vectorized source code for a typical motion estimation function with vector length 4.

```
for (j = 0; j < max_y_search; j++)
  for (i = 0; i < max_x_search; i++)
    for (y = 0; y < macro_ block_height; y++)
      for (x = 0; x < macro_ block_width; x = x + 4)
        SAD[i][j] += VectReduction(prevFrame[i+x][j+y] - currFrame[x][y]);
```

VectorReduction takes two vectors of length 4 and produces a scalar value. Note that *x* index is increased by 4 because the next vector is 4 elements away.

Figure 4 shows a simple example of a typical vectorized method for motion estimation using 2x2 macroblock and 3x3 search area. In this example, we assume that the register contains a 2x2 block of the image. In most cases, elements in a 2x2 blocks are not consecutive in the memory, hence special instruction(s) is (are) needed to read them into a register.

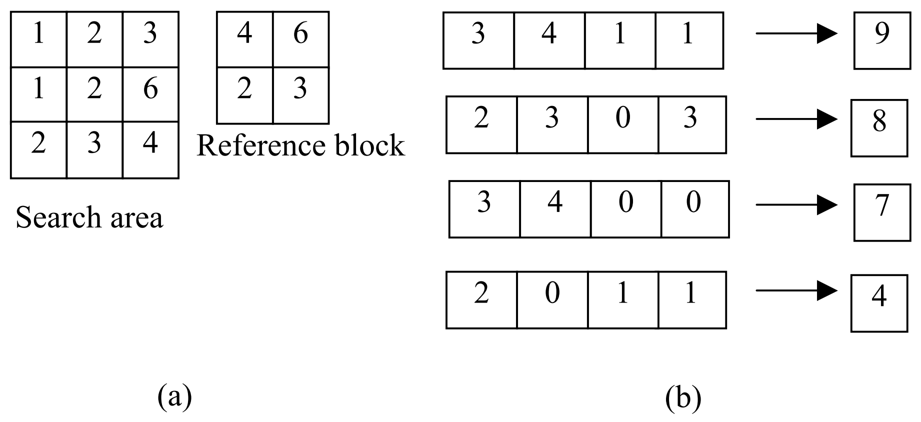


Figure 4. *(a) Search area and reference block, (b) Vector registers containing the absolute differences and scalar results from applying reduction instruction on vector registers.*

The procedure for an exhaustive search is as follows. If one places the reference block on top of the search area, as shown in Fig. 4(a), and starts sliding from left to right and top to bottom. There are 4 possible positions for the reference block. In each possible position, subtracting the corresponding elements of the reference block and the search area block, and taking the absolute value of the subtraction, results in four 4-dimensional vectors as shown in Figure 4(b). In order to find out the minimum, we need to use four *reduction* operations to obtain four scalar values and choose the minimum value of the four, which in this case happens to be 4.

The advantage of vectorizing SAD is that it naturally fits into other fast motion estimation algorithms where only a small number of selected 16x16 squares in the search region are used in computing SAD. However, vectorizing SAD for VIRAM is not optimal for the exhaustive search. Our simulation shows that a straightforward implementation of exhaustive search by vectorizing SAD as shown in the above source code, results in sub-optimal performance. In our implementation of vectorizing SAD, we use vector length of 16 since each macroblock is a 16x16 square. We also use 16-bit data for vector element to avoid overflow from subtraction of 8-bit pixel values. Table 2 shows the performance of motion estimation for one frame in both QCIF and CIF sequence.

Size	Cycles	Giga-operation/sec	Giga-operation/sec Peak	% Peak
QCIF	$2.3x10^6$	1.63	6.4	25.5%
CIF	$1.0x10^7$	1.64	6.4	25.6%

Table 2. *Performance of motion estimation using a typical algorithm of vectorizing SAD.*

5.1.2 **Discussion**

As indicated in Table 2, VIRAM operates only at a quarter of its capability in computing motion estimation. There are two factors that cause VIRAM to perform poorly on motion estimation using vectorizing SAD algorithm.

1. Vector length is too short which results in only one active vector arithmetic functional unit (VAFU) at a time.
2. Reduction happens inside the inner loop is expensive as it has long latency, breaks the vector pipeline, and reduces opportunities for optimization.

Why short vector length results in less efficient usage of hardware?

VIRAM can only issue one instruction per cycle and each VAFU can compute 256 bits per cycle. Because the vector contains only 16 16-bit elements (256 bits), each instruction is completed in one cycle using only one VAFU, leaving the other VAFU idle. If the vector length is longer then one VAFU won't finish one instruction in one cycle. Therefore, the next instruction will be assigned to the second VAFU. Now, we have two VAFUs working at the same time, hence better hardware usage.

Why is reduction inside the inner loop expensive?

We use 32 vector registers, each has 16 elements, to store the current macroblock and the search block in previous frame. To find the SAD between the two blocks, we need to do 16 reductions on 16 registers. In H.263, by default, there are 31x31 possible search blocks in previous frame, hence we need 31x31x16 *reduction* operations to find out the minimum SAD for one block. VIRAM has *no reduction* instruction, but it has a way of emulating *reduction* instruction by using simpler instructions. However, this incurs high latency of *reduction* operation. In addition, the reduction also reduces the vector length, which affects subsequent vector instructions. Hence, vector instructions cannot be moved before or after the *reduction* operation during hand optimization. Alternatively, we propose a different approach to implement exhaustive search algorithm that uses long vector and few *reduction* operations.

5.2 Motion Estimation Algorithm by Vectorizing Data across Macroblocks.

We apply the new algorithm to the same simple example in the previous section, and show that only one *min reduction* operation is needed as compared to four with previous algorithm. *Min reduction* operation is the same as *reduction* operation except it returns the smallest element in a vector. We first describe the algorithm and then present the VIRAM architecture features that enable the algorithm.

5.2.1 Description

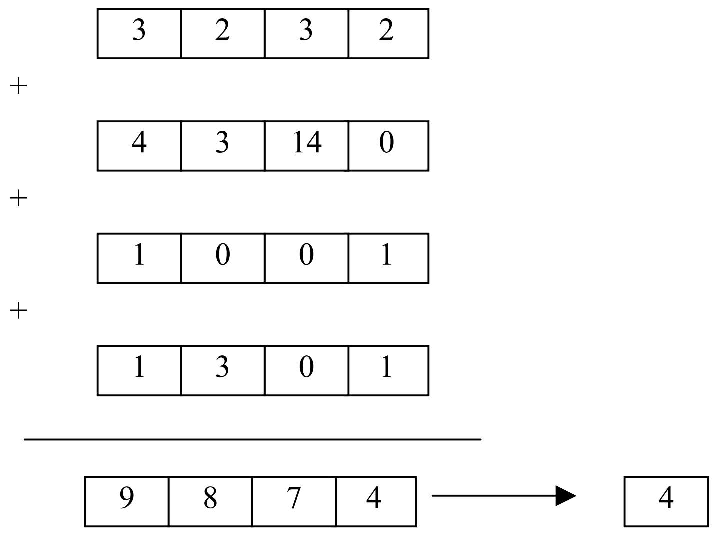


Figure 5. *Different implementation of motion estimation to use fewer reduction instruction.*

The method is as follows. *Step one*: Subtract each entry in the reference block from all possible overlapped entries of the search area, and take the absolute value of the subtraction. For example, the first entry (0,0) of the reference block will overlap with entries (0,0), (0,1), (1,0), (1,1) of the search area, and the resulting vector is (3,2,3,2). Since the reference block has 4 elements in this example, we will have 4 resulting vectors as shown in Figure 5. *Step two:* Add all resulting vectors to obtain a final vector (9,8,7,4). Each element in this vector is the SAD at a particular position in the search area. *Step three:* Use only one *min reduction* operation on the final vector to obtain the minimum SAD.

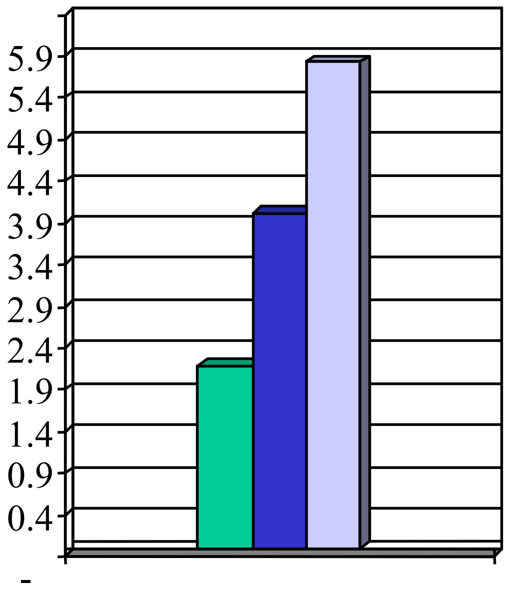
5.2.2 Discussion

To implement the above algorithm efficiently, the scalar and vector subtraction instruction is needed to perform the *step one* in the example. Unlike other architectures such as MMX, VIRAM has a wide range of arithmetic scalar and vector instructions that take a scalar operand and a vector operand to produce a vector result. Next, to store many possible overlapped entries of the search area in a few vector registers to perform the subtraction in *step one*, the vector registers have to be long. While most existing architectures have 64 to 128 bits vector registers, hence can only store 4 to 8 16-bit values, VIRAM's registers are 2048-bit long that can store up to 128 16-bit values. As such, two VIRAM vector registers are enough to hold 16x16 overlapped search area. In addition, VIRAM provides the *indexed load* instruction to read the overlapped search area, which is usually a block in the image, into a vector register. Other DSPs such as the TriMedia [17] do not have the ability to load a block of the image into a vector register in one instruction. In *step 3*, VIRAM provides an efficient way to do min reduction on a N-dimensional vector with log(N) complexity. Note that we need only two *min reduction* on 128-dimensional vectors to calculate the minimum SAD for a macroblock as compared to 16x31x31 *reduction* operation on 16-dimensional vectors in the previous algorithm.

Tables 2 shows the performance of three versions of VIRAM vs. Pentium II MMX. The number in the parentheses represents the speed-up factor of VIRAM over MMX. On the Pentium MMX, the measuring time is multiplied by the clock rate to obtain the number of cycles, and the measurement is done using SAD routine provided by Intel Corp [8]. On VIRAM, we use the cycle-accurate performance simulator. Figure 6 shows the efficiency of hardware usage on motion estimation. VIRAM-1 is the current design that can generate four addresses per cycle, VIRAM-2 design can generate eight addresses per cycle, and VIRAM-3 can generate 16 addresses per cycle for *indexed* *loads*. As seen in Table 2, VIRAM-1 performance is much worse than VIRAM-2 and VIRAM-3 because of the stalls in address generation units by *indexed loads*. Address generation stalls happen when address computation are not fast enough for the vector arithmetic functional unit. Still, VIRAM-1 outperforms MMX by a factor of 4.6 in number of cycles.

Size	VIRAM-1	VIRAM-2	VIRAM-3	MMX
QCIF	$7.1\text{x}10^6$ (4.6x)	$3.9\text{x}10^6$ (8.4x)	$2.7\text{x}10^6$ (12.2x)	$3.3\text{x}10^7$
CIF	$2.8\text{x}10^7$ (5.0x)	$1.6\text{x}10^7$ (8.7x)	$1.1\text{x}10^7$ (12.7x)	$1.4\text{x}10^8$

Table 3. *Cycles/frame for the exhaustive search*



Addr/cycle	GOPS	Peak	% of Peak
4	2.2	6.4	34.1
8	4.02	6.4	62.8
16	5.83	6.4	91.1

Index load unit can generate:

🟩 4 addresses/cycle

🟦 8 addresses/cycle

◻ 16 addresses/cycle

Figure 6. *Efficiency throughput for VIRAM with 4, 8, and 16 addresses/cycle.*

6 Discrete Cosine Transform

6.1.1 Description

The discrete cosine transform maps the pixel values from spatial domain into the frequency domain for energy compression. The two-dimensional forward DCT of NxN pixels *f(x,y)* is given by:

$$F(u,v) = \frac{2}{N} k_u k_v \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left(\frac{(2x+1)u\pi}{2N}\right)\cos\left(\frac{(2y+1)v\pi}{2N}\right)$$

where $k_i = 1/\sqrt{2}$ for i = 0 and $k_i = 1$ otherwise.

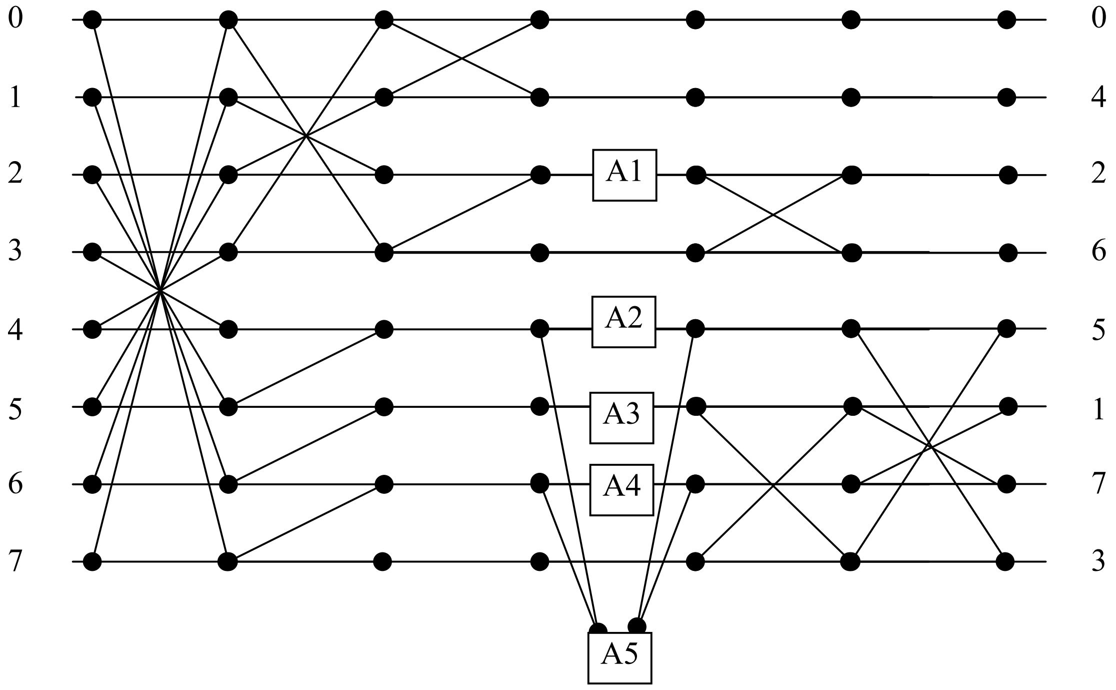


Figure 7. *Flow graph for 1-D DCT algorithm by Arai, Agui, and Nakajima.*

There are several fast methods for computing the DCT. In this thesis, we consider only two popular algorithms for computing DCT: LLM [14] and AAN [1]. While the original LLM algorithm uses 11 multiplications and 29 additions, we implement the alternate LLM [14], which uses 12 multiplications and 32 additions. The advantage of this method is that no data path contains more than one multiplication. This allows a simple and accurate implementation in scaled fixed-point arithmetic, with a minimal number of shifts. We also use 32-bit data to comply with the MPEG standard. For the AAN algorithm, we use 16-bit data to speed up the computation at the expense of accuracy. A typical fast 1-D DCT algorithm has a similar flow graph as shown in Figure 7. The flow graph exhibits a straightforward vectorization of the algorithm.

In both algorithms, we first take a 1-dimensional DCT along the column, and then take another 1-dimensional DCT along the row. To do the DCT along the column, we use *unit strided load* to read the first 8 rows of the image into 8 vector registers. We then apply the normal DCT operations across the vector registers as though they are scalar values*,* and use *unit strided store* to write back the results of column DCT. This process repeats for all the rows of the image. Next, to take the DCT along the row, we apply the same method as above except using *strided load and store* to work with the columns of the image.

6.1.2 Discussion

Unlike other architectures, VIRAM has high memory bandwidth due to its embedded DRAM technology that allows efficient *strided load and stores* [13].
However, our simulations show that the DCT performance of VIRAM degrades due to the address conflicts and address generation stalls.

Address conflict

Address conflict happens when there is an access to the same memory bank while the previous access has not been completed. In a conventional system, cache miss is the primary concern for performance. However, since VIRAM has no cache, memory bank conflict can greatly reduce VIRAM speed performance. To avoid address conflicts, VIRAM memory bank is divided into many sub-banks. Figure 6 shows the average number of cycles for computing 2-dimensional DCT of an 8x8 block using QCIF image for both AAN and LLM with different number of sub-banks. As seen, the number of cycles in LLM algorithm is reduced by nearly a factor of 2, from 1 sub-bank design to 4 sub-banks design. However, we gain only 5 cycles going from 4 sub-banks design to 8 sub-banks design due to the fact that address conflict is no longer the bottleneck but the computation. In the AAN algorithm, the data in the image is only 16-bit, and therefore smaller strides in the memory are required to access the columns of image that effectively reduces the address conflicts. Hence, increasing number of sub-banks does not change the result much since the bottleneck is no longer address conflicts but the address generation stalls.

Address generation stalls

Address generation stalls happen quite often with *strided* loads and *strided* stores in computing DCT on the rows of the image. With current design of VIRAM, *strided* load can only read 4 elements per cycle while the vector arithmetic functional unit can process up to 256 bits per cycle. If 32-bit data is used as in our implementation of LLM algorithm [14], one vector arithmetic functional unit can process data twice as fast as the memory unit can read the data. This speed discrepancy leads to address generation stalls. The speed discrepancy between the vector functional arithmetic unit and the memory unit is even greater for our implementation of AAN algorithm [1] since we used only 16-bit input data.

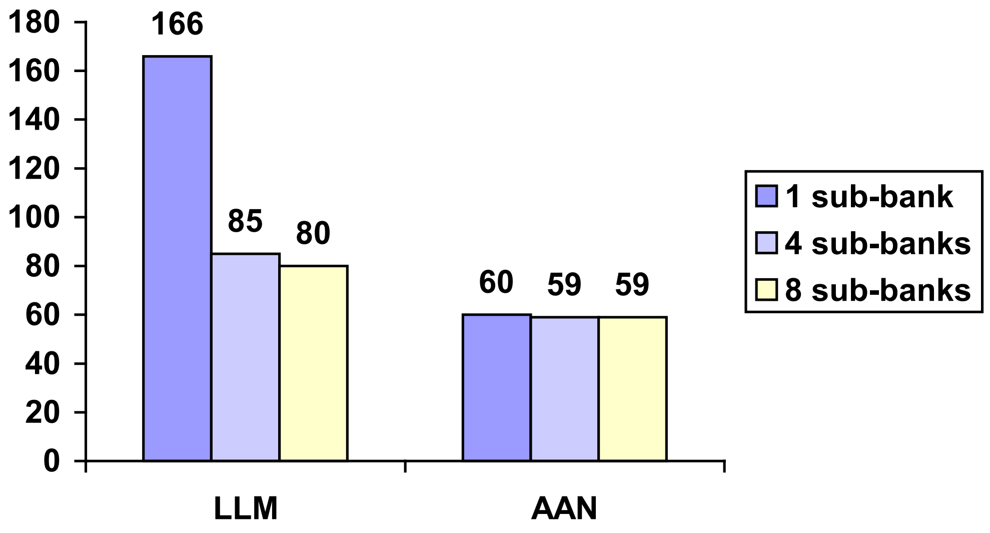


Figure 6. *Cycles/block with different numbers of sub-banks*

VIRAM (4 sub-banks, LLM)	85
CPU64 from Philips (not yet available)	56 (0.65x)
TriMedia TM-1000	160 (1.88x)
TI TMS320C62	230 (2.71x)
PowerPC with Alitvec	102 (1.20x)
HP PA-8000 with MAX2	147 (1.73x)
Intel Pentium II + MMX	500 (5.88x) [7]
NEC V 830/AV	201 (2.36x)

Table 4. *Cycles/block for different architectures*

Table 3 shows number of cycles to compute the DCT of an 8x8 block on different architectures. The number in parentheses represents the speed-up factor of VIRAM versus other architectures. Note that VIRAM outperforms other DSP chips except for the CPU64 in computing the DCT. However, the above number for CPU64 is a theoretical calculation, assumes perfect cache, and uses only 16-bit data while VIRAM number results from the actual simulation with 32-bit data and memory stalls. In addition, VIRAM, V 830, and CPU64 numbers comply with MPEG accuracy standard. For other numbers, compliance is not claimed, and hence computationally cheaper algorithms could have been used.

7 Overall Performance

Currently, our cycle-accurate simulator only emulates the vector unit of VIRAM. We do not yet have a scalar performance simulator. To measure the overall speed performance of the H.263 encoder that has both scalar and vector codes, we measure the time spent on the scalar code separately on a SGI machine, and the time spent on the vector code on the simulator. We then combine the results to get the overall performance of H.263. This method is reasonably accurate since VIRAM has the same scalar core as the SGI machine. Applying this technique to VIRAM with 4 sub-banks, we obtain the average achievable, encoding frame rate for the test sequences in Table 4 using exhaustive search for motion estimation, and LLM for DCT. As we optimize the motion estimation and the DCT, the time spent on variable length coding (VLC) becomes significant. For example, the DCT and the motion estimation of the Foreman sequence take about 42 percent of the total time while VLC and other miscellaneous operations take the other 58 percent. At present time, we have not optimized the VLC for VIRAM. As vector processing speed increases, we anticipate VLC to become the bottleneck in a typical video encoder.

Akiyo (12.95 kbit/s)	Mom (16.25 kbit/s)	Hall (20.47 kbit/s)	Foreman (65.52 kbit/s)
23.5 fps	22.7fps	22.7fps	20.9fps

Table 4. *Average encoding speed for H.263 on VIRAM.*

As seen, the achievable encoding rates are high enough to result in acceptable quality for most multimedia and communication applications.

8 Summary

In this thesis we presented an overview of VIRAM architecture, a vector microprocessor with embedded memory, optimized for multimedia applications. Although VIRAM is a general-purpose processor, its performance exceeds other DSP and multimedia extension architectures consistently by a factor of 4.6 to 8.7 in computing motion estimation and by 1.2 to 5.88 in computing discrete cosine transform. The improvement in the H.263 encoder performance is due to VIRAM's high memory bandwidth based on the embedded DRAM technology. With high memory bandwidth, VIRAM architecture is able to provide efficient *indexed* and *strided* memory operations that are well suited for memory access patterns of the discrete cosine transform and motion estimation. The long vector registers of VIRAM allow an efficient vectorized algorithm for the exhaustive search motion estimation. In addition, unlike existing multi-chip solutions, VIRAM is a one-chip solution, and as such, it results in lower power consumption, and smaller area.

9 References

[1] Arai, Agui, and Nakajima. Trans. IEICE E-71(11):1095 Pennebaker & Mitchell JPEG textbook, figure 4-8 in P&M.

[2] K. Asanovic. *Vector Microprocessors*. PhD thesis, Computer Science Division, UCB, 1998.

[3] E. BrockMeyer, et al. "Low Power Memory Storage and Transfer Organization for the MPEG-4 Full Pel Motion Estimation on a Multimedia Processor", IEEE Trans. On Multimedia, Vol. 1, No. 2, June 1999

[4] K. Diefendorff and P.Dubey. "How Multimedia Workloads Will Change Processor Design". IEEE Computer, 30(9):43-45, September 1997

[5] R. Fromm, et al. "The energy efficiency of IRAM architectures". ISCA, pages 327-337, June 1997.

[6] R. Fromm, et al. "Vector IRAM Memory Performance For Image Access Patterns". Technical Report UCB//CSD-99-1067. University of California – Berkeley-Oct 1999

[7] B. Furth, et al. "Motion Estimation Algorithms for Video Compression". ISBN 0-7293-9793-2

[8] Intel Corp. "Using MMX TM Instructions to Compute the Absolute Difference in Motion Estimation", http://developer.intel.com/drg/mmx/AppNotes/app530.htm

[9] Intel Corp. JPEG Inverse DCT and Dequantization Optimized for Pentium II Processor. http://developer.intel.com/drg/pentiumII/appnotes/886.htm

[10] J. Jain, et al. "Displacement Measurement and its Application in Interframe Image Coding", IEEE Transactions on Communications, Vol. 29, No.12, December 1981, pp. 1799-1808

[11] J. Koga, et al. “Motion Compenstated Interframe Coding for Video Conferencing”, Proceedings of the National Telecommunications Conference, 1982, pp G5.3.1-5.3.5

[12] C.E Kozyrakis and D.A. Patterson. “A New Direction in Computer Architecture Research.” IEEE Computer, 31(11):24-32, November 1998.

[13] C.E Kozyrakis. “A media-enhanced vector architecture for embedded memory systems.” Technical Report UCB//CSD-99-1059, UCB, July 1999

[14] C. Loeffler, et al. "Practical Fast 1-D DCT Algorithms with 11 Multiplications", ICASSP '89, pp. 988-991.

[15] D. Patterson, et al. “A Case for Intelligent DRAM”. IEEE Micro, April 1997

[16] Telenor, Online documents: http://spmg.ece.ubc.ca/h263plus/h263plus.html

[17] TriMedia Databook, Philips Inc