IEEE Asian Solid-State Circuits Conference
November 8-10,2010 / Beijing, China

# A 92mW 76.8GOPS Vector Matching Processor with Parallel Huffman Decoder and Query Re-ordering Buffer for Real-time Object Recognition

Seungjin Lee, Joonsoo Kwon, Jinwook Oh, Junyoung Park, and Hoi-Jun Yoo
Department of Electrical Engineering
KAIST
Daejeon, Korea
seungjin@eeinfo.kaist.ac.kr

*Abstract*—**A vector matching processor with memory bandwidth optimizations is proposed to achieve real-time matching of 128 dimensional SIFT features extracted from VGA video. The main bottleneck of feature-vector matching is the off-chip database access. We employ the locality sensitive hashing (LSH) algorithm which reduces the number of database comparisons required to match each query. In addition, database compression using Huffman coding increases the effective external bandwidth. Dedicated parallel Huffman decoder hardware ensures fast decompression of the database. A flexible query re-ordering buffer exploits overlapping accesses between queries by enabling out-of-order query processing to minimize redundant off-chip access. As a result, the 76.8 GOPS feature matching processor implemented in a 0.13um CMOS process achieves 43200 queries/second on a 100 object database while consuming peak power of 92mW.**

## I. Introduction

Today, object recognition is being used for automotive applications such as lane detection and forward collision detection as well as mobile applications such as augmented reality. These applications require real-time performance to provide fast response time and fluid visual feedback to the user. Although 15fps is generally considered acceptably fast for automotive safety applications, 25-30 fps is required for smooth perceived motion to the human eye. Low power consumption is also necessary to fit in an embedded form factor. Previous object recognition chips [1,2] employed parallel processing elements to achieve 30 fps performance. However, the limited performance of the matching stage could only support object databases of small size.

Feature matching in object recognition is basically a nearest neighbor problem. Given a query vector, the most similar vector in a database must be returned. In a brute force approach, the query vector is compared to each vector in the database and therefore the complexity for each query is O(dn), where d is the dimensionality of the vector and n is the number of vectors in the database. In scale invariant feature transform (SIFT) [3] based object recognition, the feature vector dimensionality is 128, and the database may contain on the order of 100,000 vectors. Moreover, each image frame may require up to 1,000 queries, making the time cost of brute force matching prohibitively high.

Traditional tree based search methods do not yield any performance improvement over brute force methods for vectors with dimensionality of over about 20 [4], as is the case for the 128 dimensional SIFT vectors. Approximate methods, such as the locality sensitive hashing (LSH) [4], can dramatically reduce the computational complexity by allowing for a controlled amount of error $\epsilon$. In LSH, a hash function confines each query search to a "bucket" which contains a relatively small number of database vectors that are likely to be close to the query vector. As a result, the time complexity can be reduced to $O(dn1/(1+\epsilon))$, which can be orders of magnitude faster than O(dn) of brute force matching for large databases.

Even with LSH, however, the external memory access is still a bottleneck of feature matching performance. Due to its large size, the database must be stored on off-chip memory. Taking a single comparison between a query vector and a database vector as an example, the actual distance calculation between the two vectors is easily accelerated through data parallelism and pipelining, and it is realistic to achieve 1 comparison/cycle throughput without excessive hardware cost. However, fetching a single vector from an external database will take 32 cycles even on an idealized 32bit bus.

In this paper, we propose a vector matching processor with three key features for achieving real-time feature matching on a highly integrated object recognition SoC [5]. First, LSH optimized hardware architecture reduces the computational complexity. Second, a parallel Huffman decoder operating at bus speed increases the effective external database vector fetch rate. Third, a query re-ordering buffer maximizes data reuse in an on-chip cache to minimize the number of off-chip fetches.

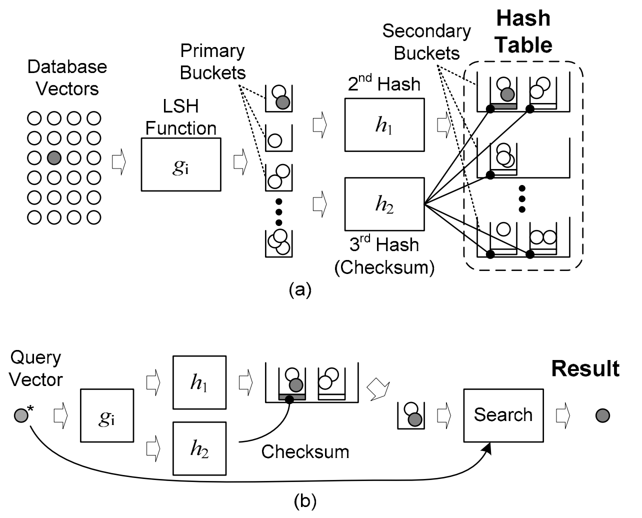


Figure 1. (a) Database preprocessing for LSH (b) vector querying using LSH.

## II. Algorithm Analysis

There have been several previous works on approximate nearest neighbor search for high-dimensional vectors. The best-bin-first (BBF) method [6] is best known for its use with SIFT feature vectors. It is a modification to the kd-tree exact search method. BBF and LSH are known to perform comparably from empirical studies [7] on sequential hardware. However, BBF requires sequential transversal of the kd-tree, while LSH hashing can be performed in parallel.

### A. *Locality Sensitive Hashing (LSH)*

Feature matching using the LSH algorithm is outlined in fig. 1. In order to perform matching with LSH, the feature database must first be organized into a hash table, as shown in fig.1(a). The main hashing is done by the LSH function denoted $g_i$. The role of $g_i$ is to group vectors that are close together into the same buckets. Applying $g_i$, however, results in a very large number of buckets making such a hash table impossible to manage. Thus, a second hash function, denoted $h_1$, is used to assign the "primary buckets" to a predetermined number of "secondary buckets". The second hashing inevitably results in collisions, so a third hash function, denoted $h_2$, assigns a checksum to each vector that can be used to distinguish vectors originating from different primary buckets.

The query matching process, shown in fig. 1(b), is nearly identical to database preparation. The query vector is first hashed using $g_i$. The result of this is then hashed by $h_1$ to obtain the corresponding secondary bucket from the hash table. Since this bucket may contain vectors from several primary buckets, $h_2$ is applied to select only those vectors that correspond to the correct primary bucket. The selected vectors are then compared to the query vector to obtain the approximate nearest neighbor match result.

The LSH function $g_i$ is usually randomly generated. In order to improve the matching accuracy, multiple LSH functions are employed. In this work, we use 20 LSH functions.

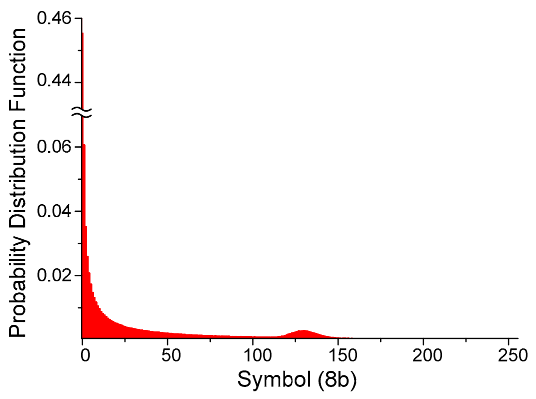


Figure 2. PDF of symbols in SIFT descriptor vectors.

### B. *Database Compression using Huffman Coding*

The value distribution of SIFT feature vectors in the database, shown in fig. 2, is well suited for compression by Huffman coding [8]. Huffman coding is a variable code length compression method that represents frequently occurring symbols using shorter codes to reduce overall data size. In the example, which is the distribution for a 40,000 vector database, 0 accounts for 45.6% of all symbol occurrences. With Huffman coding, the symbol 0, which was originally 8 bits, is represented by the binary code '0', which only takes up 1 bit. A brief example of the code mappings is shown in table 1.

TABLE I. Huffman Code Mapping

| Symbol | Frequency | Code | Code Length |
|---|---|---|---|
| 0 | 0.46 | 0 | 1 |
| 1 | 0.061 | 1010 | 4 |
| 2 | 0.035 | 11010 | 5 |
| 3 | 0.026 | 10001 | 5 |
| 4 | 0.021 | 111010 | 6 |
| ... | ... | ... | ... |

### C. *Temporal Locality*

During actual object recognition, vector queries that are close together in time tend to be similar to each other. This could be due to repeating patterns or symmetric textures in an image (since SIFT is invariant to rotation, symmetric textures

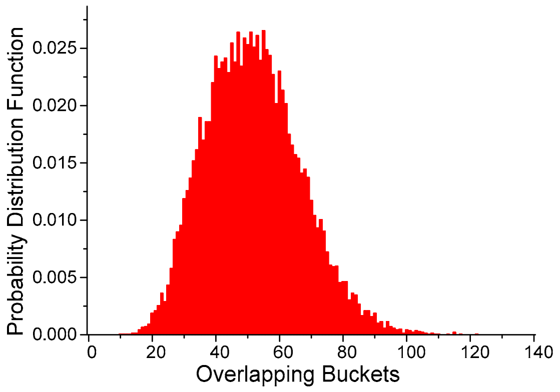


Figure 3. Overlapping buckets among 8 consecutive queries.

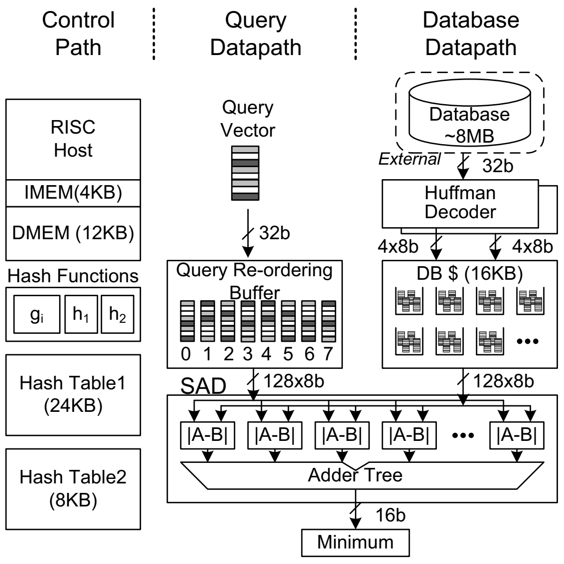


Figure 4. Top architecture of the vector matching processor.

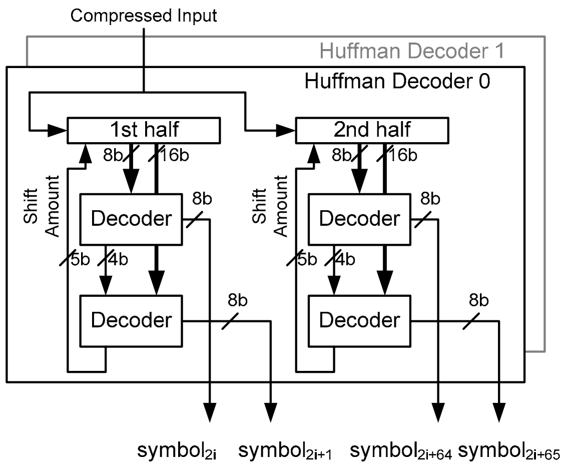


Figure 5. Parallel Huffman decoder.

will have similar descriptor vectors). Fig. 3 shows the probability distribution function for overlapping buckets for 8 consecutive queries. As described earlier, each query is hashed using 20 hash functions. Therefore, normally 160 buckets will need to be read from the external memory. However, if the overlapping buckets can be reused, a considerable amount of off-chip access can be reduced.

## III. ARCHITECTURE

The top architecture of the vector matching processor is shown in fig. 3. It is composed of 4 main parts. First, the programmable RISC host is responsible for the overall control. Second, a configurable dedicated hash function and on-chip hash tables provide the necessary functions for the LSH algorithm. Third, parallel Huffman decoders capable of decoding 8 symbols per cycle provide high effective bandwidth to fill the DB cache. Fourth, a 16KB database cache and an 8 entry query buffer enables high performance.

### *A. Hash Function and Hash Tables*

The hash function contains separate hardware for the $g_i$, $h_1$, and $h_2$ functions. Each provide single cycle latency for low overhead. For this, the $g_i$ LSH function is connected directly to the 128B interface of the query buffer. The $h_1$ and $h_2$ functions are simply implemented as modulus operations. The number of secondary buckets in the hash table is set to 149.

The hash tables contain the links to actual vectors that are located on the off-chip database. A double indexing data structure is adopted so that each (fixed size) entry in hash table2, shown in fig. 4, points to the actual variable sized bucket stored in hash table1. Each entry of hash table1 contains information on each vector such as the compression ratio and address offset. Since low latency is crucial, the hash tables are stored in dedicated on-chip memories.

### *B. Parallel Huffman Decoder*

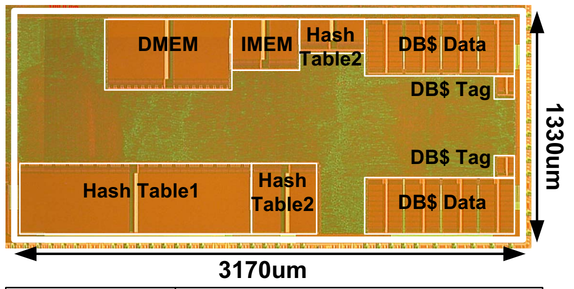
Due to variable length codes, Huffman decoding is a fundamentally sequential task. In order to accelerate this, the input stream must be divided into predefined blocks. We enforce three fixed sizes for compressed vectors: 64B, 96B, and 128B corresponding to 50%, 25% and 0% compression ratio. This, however, comes at the price of compression performance. For comparison, direct application of the codes in table 1 without blocks can achieve 44% reduction in the database size. With blocks, any space between the end of a vector and the beginning of the next vector is wasted. As a result an overall compression ratio of just over 30% was observed in this case.

The parallel Huffman decoder is shown in fig. 5. Two vectors are processed in parallel. Each compressed vector is again divided into equal halves for parallel processing. As long as the code lengths are all equal to or smaller than 8, all four decoders are able to produce 1 symbol in a cycle. As a result, the maximum throughput of the two decoders is 8 symbols per cycle.

### *C. Query Re-ordering Buffer*

As shown in fig. 3, consecutive queries exhibit temporal locality in that they tend to yield similar hash results. The query re-ordering buffer and the database cache shown in fig. 4 can exploit this fact to eliminate unneeded off-chip accesses. Since each of the query vectors in the re-ordering buffer are connected directly to the sum of absolute differences (SAD) unit by a 128B datapath, SAD calculation can be performed anytime a required bucket is available in the database cache without overhead. Therefore, the queries are processed out-of-order, bucket-by-bucket based on data availability. Meanwhile, the RISC host program executes an algorithm that encourages processing of queries that are near completion so as to encourage introduction of new sources of data overlap.

The SAD unit is capable of 1 128B SAD operation per cycle. The SAD performance was intentionally made unproportionately high since it only requires relatively small area.



| Technology | 0.13um 1P8M Logic CMOS | |
|---|---|---|
| Area | 3170 um x 1330 um | |
| Gates / SRAM | 255K Gates / 64 KB | |
| Power Supply | 0.65 ~ 1.2 V | |
| Frequency | 50 ~ 200MHz (90FO4) | |
| Peak Performance | Operations | 76.8GOPS |
| | Matching | 43200 queries/s |
| Power | Peak | 92mW @ 200MHz |
| | Average | 65mW @ DVFS |

Figure 6. Chip photograph and summary.

## IV. Implementation

The proposed processor, shown in fig. 6, is implemented in a 0.13um CMOS process and contains 255K gates and 64KB of on-chip SRAM. It is part of an object recognition SoC [5] which is 5x10mm$^2$. Communication with the rest of the chip is achieved through on on-chip network interface which provides 640MB/s of bandwidth in either direction. The bottleneck in database access was the DDR DRAM, which maxed at about 200MB/s. The processor was designed to operate over a wide range of voltage/frequency combinations to enable low power operation during light loads through DVFS. The minimum voltage/frequency combination of 0.65V/50MHz consumes only 14% of the peak power while providing 25% of the peak performance. As a result, a low average power of 65mW can be achieved for low loads.

Experimental tests of the processor were carried out using the setup shown in fig. 7. A database containing 42071 features from 100 objects of the COIL-100 object set was constructed. Matching was performed successfully on test images like the ones shown in fig. 8 at a rate of 43200 queries per second. Comparison with PC simulation using brute force matching revealed less than 4% mismatches for overall queries, which is negligible considering the performance increase.

## V. Conclusion

A high performance vector matching processor is implemented to enable robust object recognition with large databases in an embedded system. Locality sensitive hashing is employed to minimize the computational complexity of matching. Moreover, database compression through Huffman coding and database reuse through a query re-ordering buffer alleviate the off-chip bandwidth bottleneck to improve matching performance. As a result, the vector matching processor is capable of processing 43200 queries/s for a 42071 feature database of 100 objects, enabling smart vision applications on embedded and mobile platforms.

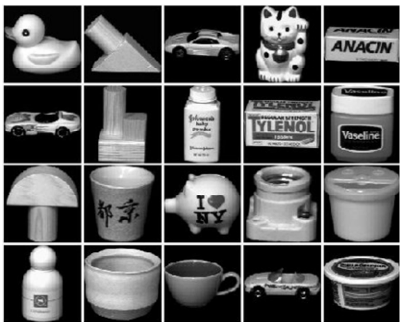


Figure 7. Example objects from the COIL-100 database.

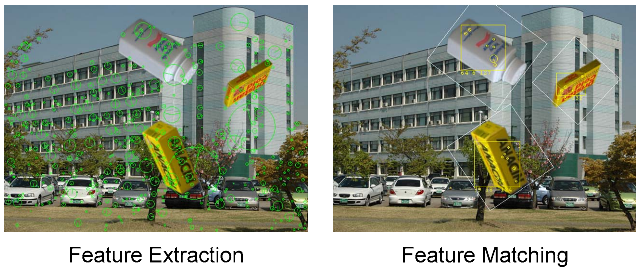


Figure 8. Vector matching results.

## References

[1] K. Kim, et al., "A 125 GOPS 583 mW Network-on-Chip Based Parallel Processor With Bio-Inspired Visual Attention Engine," IEEE Journal of Solid-State Circuits, vol. 44, no. 1, pp. 136-147, 2009.

[2] J.-Y. Kim, et al., "A 201.4GOPS 496mW Real-Time Multi-Object Recognition Processor with Bio-Inspired Neural Perception Engine," IEEE Journal of Solid-State Circuits, vol. 45, no. 1, pp. 32-45, 2010.

[3] D.G. Lowe, "Distinctive image features from scale-invariant keypoints," Intl. J. Comp. Vis., vol. 60, no. 2, pp. 91-110, 2004.

[4] A. Gionis, P. Indyk, R. Motwani, "Similarity Search in High Dimensions via Hashing," Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[5] S. Lee, et al., "A 345mW Heterogeneous Many-Core Processor with an Intelligent Inference Engine for Robust Object Recognition," IEEE ISSCC 2010 Dig. Tech. Papers, pp.332-333.

[6] J.S. Beis, and D.G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition.

[7] A. Auclair, L.D. Cohen, and N. Vincent, "How to Use SIFT Vectors to Analyze an Image with Database Templates," Lecture Notes in Computer Science, vol. 4918, pp. 224-236, 2008.

[8] D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., September 1952, pp 1098–1102.