A High Performance Systolic Architecture for k-NN Classification

Kevin R. Townsend, Phillip Jones, Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
{ktown, phjones, zambreno}@iastate.edu

***Abstract*—This paper describes the architecture of the winning entry to the 2014 Memocode Design Contest, in the maximum performance category. This year's Memocode design contest asks contestants to find the 10 nearest neighbors between 1,000 testing points and 10,000,000 training points. Instead of using Euclidean distance, the contest uses Mahalanobis distance. The contest has 2 awards: the maximum performance award and the cost adjusted performance award.**

Our implementation uses a brute force approach that calculates the distance between every testing point to every training point. We use the Convey HC-2ex, a FPGA-based platform. However, the theory applies to software implementations as well. At the time of publication, our runtime is 0.54 seconds.

I. Introduction

The problem of the 2014 Memocode design contest is to find the k nearest neighbors among a training set of M train vectors for N test vectors. Instead of using the Euclidean distance, $\sqrt{(x-y)^t(x-y)}$, where x is a train point and y is a testing point, the competition asks us to find the Mahalanobis distance, $\sqrt{(x-y)^tS^{-1}(x-y)}$. S^{-1}, the inverse covariance matrix, is given to us. Since the square root function is an increasing function, it can be dropped and the nearest neighbors will be the same, $(x-y)^tS^{-1}(x-y)$. Each vector is composed of 32 12-bit values. The large dataset, which was used to judge performance, consisted of 1,000 test vectors and 10,000,000 train vectors.

Mahalanobis distance works well for some applications [1]–[3]. This distance metric normalizes the dimensions to standard deviations and also takes into account correlations. For example, if the objective is to find hospital patients similar to a given patient, then the patients' information can be broken down into different dimensions representing age, gender, number of hospital visits, blood pressure, etc. The euclidean distance between two patients would be meaningless. We want a unitless measurement, and that is what the Mahalanobis distance achieves.

Our solution is brute force in the sense that the distance between every test vector / train vector pair is calculated. In this contest, this means 10,000,000,000 distance calculations are required. The bottleneck of our design was the number of multipliers we could fit on the chip. To fit 32 processing elements (1024 multipliers), we used lookup table (LUT) multipliers as well as multiplier (DSP) blocks. As of the writing of this paper, our runtime is 0.54 seconds.

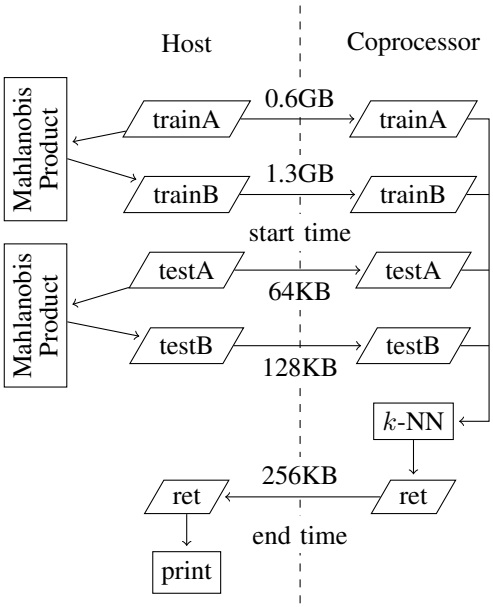


Figure 1: As the dataflow shows, the timed data movement is small and takes a short amount of time.

We describe the specifics of our algorithm in Section II. In Section III, the platform used for implementation is introduced. Section IV describes the high-level systolic architecture of our solution. In Section V, we present architectural details of an individual processing element. In Section VI, we discuss results. Implementation alternatives are discussed in Section VII. We conclude the paper in Section VIII.

II. Algorithm

Our algorithm is a brute force approach that calculates all 10,000,000,000 distances. In this way our algorithm is similar to the naïve reference implementation provided by the contest. The primary difference is in how we calculate the Mahalanobis distance. Instead of computing $(x-y)^tS^{-1}(x-y)$, we use the distributed property of matrices and compute $(x-y)^t(S^{-1}x-S^{-1}y)$. This creates 4 arrays of vectors: x, y, $S^{-1}x$, and $S^{-1}y$. We call these arrays trainA, testA, trainB, and testB respectfully.

Figure 1 shows the dataflow of our approach. Since preprocessing is allowed, we calculate trainB and move trainA

and trainB to the coprocessor (Section III) before starting the timer. testA and testB are comparatively smaller, and it takes little time to calculate testB and move both arrays to the coprocessor. We perform the dot product and sorting operation on the coprocessor. Performing sorting on the coprocessor reduces the data transferred back to the CPU board from 80GB to 256KB.

III. Convey HC-2ex

We chose the Convey HC-2ex [4] as our computing platform (Figure 2). For the contest, we estimated the machine cost as $100,000. This gave a normalized runtime of 1,486 $hours. The HC-2ex is an Intel server with an FPGA coprocessor board. The coprocessor has 4 Xilinx Virtex-6 LX760 FPGAs [5], 8 memory controllers, and is connected to a host (CPU) board by PCIe. Our design uses 311,798/474,240 (65%), 433,133/948,480 (45%), 275/720 (38%), and 640/864 (74%) of the LUTs, registers, BlockRAMs and DSP blocks respectfully on each FPGA. The clock frequency is 150 Mhz, 38% of the maximum frequency, 400 Mhz. We use a read-first RAM that has a max frequency of 400 Mhz. The DSP blocks have a maximum frequency of 450Mhz.

IV. High Level Implementation

From the high level, we can see the brute force solution requires $O(M \times N)$ computation, 10,000,000,000 Mahalanobis products. However, there is only $O(M + N)$ data, 10,001,000 vectors. The computation to data ratio suggests that the design will be compute bound, and indeed, our final design is compute bound.

The compute bound does not end the story. 1,000 test vectors turns out to be quite small. To evenly split the vectors among the processing elements we add 24 empty vectors for a total of 1,024 vectors. We assign 8 test vectors to each of the 128 processing elements.

The more test vectors each PE handles, the lower the memory bandwidth needed to load the training vectors. One vector pair, x and $S^{-1}x$, is 192 bytes, when values in x and $S^{-1}x$ are aligned to the 2 and 4 byte boundary respectfully. So the bandwidth required is 192 bytes per 8 clock cycles or 24 bytes per clock cycle. This equals 3.6 GB/s, which is less than the 19 GB/s memory bandwidth limit of each FPGA. To ensure enough memory bandwidth, we use 4 of the 16, 150 Mhz 64 bit memory ports. The HC-2 doubles the number of memory ports by clocking the memory ports at 300Mhz.

However, routing 192 wires from the memory controllers to all the processing elements is difficult for place and route tools. Instead, we create a systolic array architecture (Figure 3). The control signals and the 192 bit data signal are setup in the systolic array.

V. Low Level Implementation

Each processing element consists of 5 sub components, as illustrated in Figure 4. The first is the Buffer. The buffer either buffers a vector pair from testA and testB, or trainA and trainB. The Buffer shifts 192 bits every clock cycle from the "Data in" line. The Test Cache receives data from the Buffer once it contains a complete vector pair. After the test vectors are loaded, the Train Buffer receives data from the Buffer. Unless a stall occurs, the Train Buffer receives a new vector pair every 8 clock cycles.

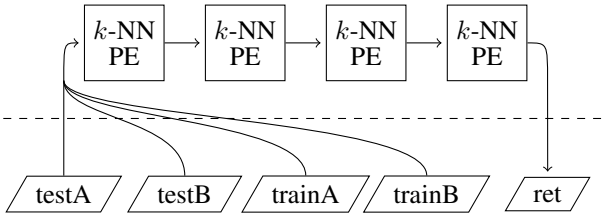


Figure 3: Systolic array view of k-NN PEs

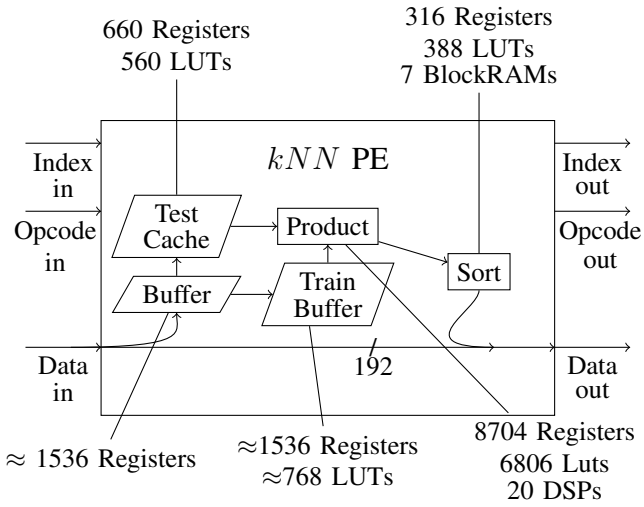


Figure 4: A single kNN PE uses 12208 registers, 8276 LUTs, 7 BlockRAMs, and 20 DSPs. As the diagram shows, the product block consumes most of these resources. We do not have exact numbers for the Buffer and Train Buffer because in our hardware description they are not separate components.

The Product block contains most of the resources. This block receives data from the Test Cache and the Train Buffer. The Test Cache increments its address every clock cycle so 8 products are calculated from each Train vector that reaches the Train Buffer.

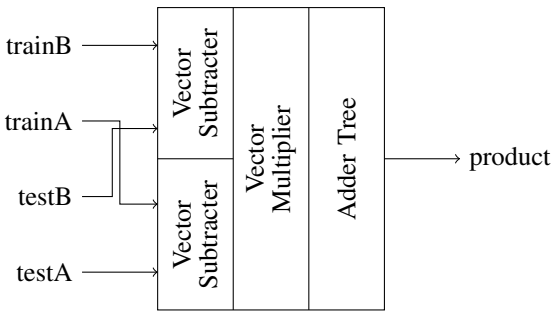


Figure 5: The dot product pipeline

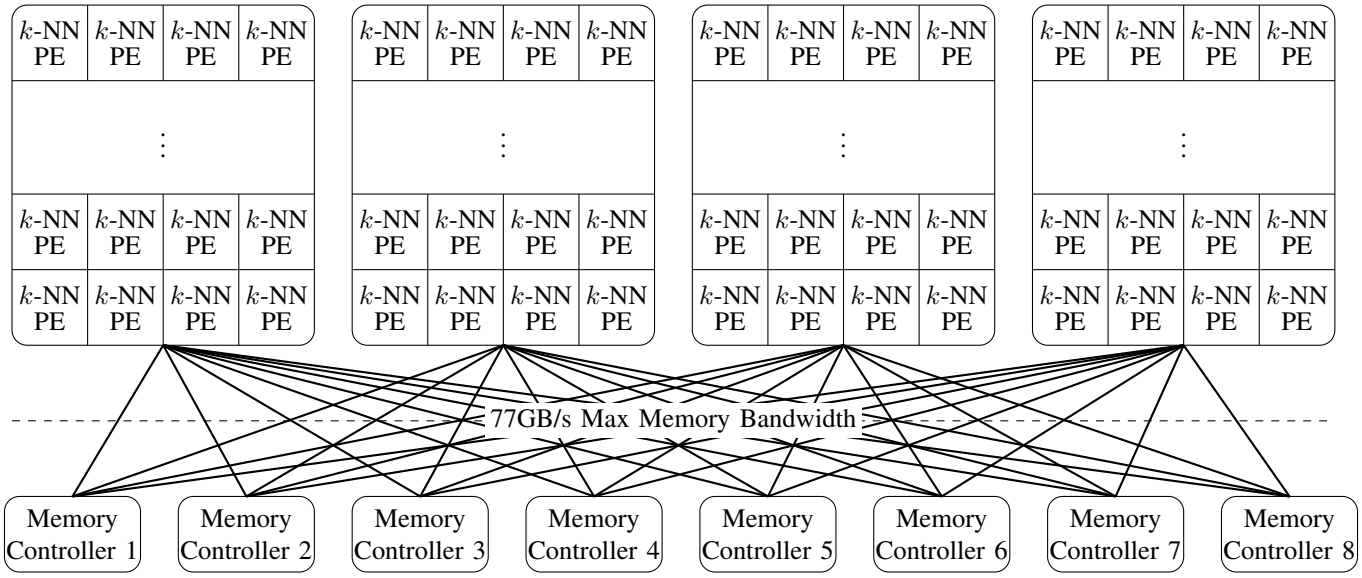


Figure 2: The k-NN implementation on the Convey HC-2ex coprocessor: 4 Virtex-6 LX760 FPGAs tiled with 32 k-NN processing elements each.

The product module (Figure 5) first calculates the differences between each value in the train and test vector pairs. The product module requires 32 multiplications (13bit by 25bit multiplication), which when efficiently implemented require 1 DSP block or 385 LUTs each. The accumulation is done by implementing an adder tree.

The Sort block keeps track of the 16 nearest neighbors for each of the 8 test vectors. The (index, value) pairs are stored contiguously in a single RAM.

For our sorter (Figure 6), we implemented insertion sort. Each insertion takes 20 cycles, however, on average, the sorter must process one value each clock cycle or the pipeline will stall. A bouncer module stores the last element in each array so that new elements larger than this value will not be queued for insertion.

We illustrate the sorter block with an example (Figure 6). In the example, we insert the number 13 into an array of 4 values. To simplify the diagram, we reduced the nearest neighbors from 16 to 4 and removed the train vector indices from the diagram.

VI. Results

The 0.54 second runtime measured equals the theoretical runtime. Our theoretical runtime can be calculated with the following formula:

$$\frac{(test\_vectors\_per\_PE) \times M \times (Number\_of\_runs)}{Clock\_frequency} \quad (1)$$

Since our design computes 1024 test vectors each run, only one run is needed to compute the 1000 test vectors in the large dataset. The FPGA is clocked at 150Mhz so the result is:

$$\frac{8 \times 10000000 \times 1}{150000000} = \frac{8}{15} = 0.54 \quad (2)$$

VII. Other implementations

We explored other implementations. At the end of the competition, our design had 64 processing elements instead of the current 128 and therefore had a runtime of 1.07 seconds.

Other changes could push performance further. For example, using CPUs or other platforms, using floating point numbers, and using approximate solutions.

A. CPU

Other implementations can also follow many of our steps. We implemented a CPU version of this approach with a runtime of 60 seconds. We chose a 2, X5650 Intel CPU platform. Our calculations show that without SIMD instructions the maximum performance is 40 seconds on this platform.

Similar to our FPGA implementation, we gave each thread 8 test vectors. We structured the nested "for" loops so the training vectors are iterated on the outer loop. If the training vector was on the inner loop, then the vectors would be loaded multiple times and require more memory bandwidth. When we switched the loops, the runtime increased to 120 seconds.

B. Floating Point

It may seem counter intuitive to use floating point numbers over integer numbers. However, CPUs often have better performance for 4-byte floating point numebers than 8-byte integers because of SIMD instructions on 4-byte values. FPGAs similarly get better performance. For example, 4-byte floating point multiplication uses less resources than 8-byte integer multiplication.

In the context of this competition, the 13-bit and 25-bit integers would be replaced with smaller floating point numbers. Using floating point numbers does cause a precision loss. In the benchmarks, the nearest neighbors were all usually more than 1% different. This can be used to our advantage

by first calculating the nearest neighbors using floating point values and then going back and calculating the exact distances and fixing any mis-orderings.

This will not work if many training points are very close to the same distance to the testing points.

C. Approximate Nearest Neighbor

Solutions that are more efficient than brute force are attractive. However, they often miss some of the nearest neighbors.

Some implementations of k dimension trees (KD-tree) [6] do not miss values and efficiently calculate a nearest neighbor. Calculating multiple nearest neighbors causes complications. KD-trees also do not handle high dimensional data well. For this reason we did not pursue this route.

KD-trees can also find approximate answers faster [7]. For example, correctly finding 9 of the 10 nearest neighbors. Other methods for approximate neareast neighbors (ANN) also exist, such as k-NN graph methods [8] and hashing [9].

VIII. Conclusion

In summary, we chose a brute force implementation to accelerate the 2014 Memocode design contest problem. Current research fails to show improvements using non-brute force implementations. By using an FPGA, we were able to compute approximately 2.4 trillion integer operations per second. This allowed us to achieve a runtime of 0.54 seconds and consequently win the maximum performance award.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under the awards CNS-1116810 and CCF-1149539.

References

[1] H. Ya-juan, H. Zhen, and S. Guo-fang, "Research for multidimensional systems diagnostic analysis based on improved mahalanobis distance," in *Proceedings of the IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, Oct. 2009, pp. 213–217.

[2] G. Verdier and A. Ferreira, "Adaptive mahalanobis distance and k-nearest neighbor rule for fault detection in semiconductor manufacturing," *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, no. 1, pp. 59–68, Feb. 2011.

[3] H. Wang, Y. Gao, and C. Zhang, "Multi-class support vector machines based on the mahalanobis distance," in *Proceedings of the IEEE International Conference on Machine Learning and Cybernetics (ICMLC)*, vol. 2, Jul. 2011, pp. 757–762.

[4] *Convey Reference Manual*, 1st ed., Convey, Richardson, TX, May 2012.

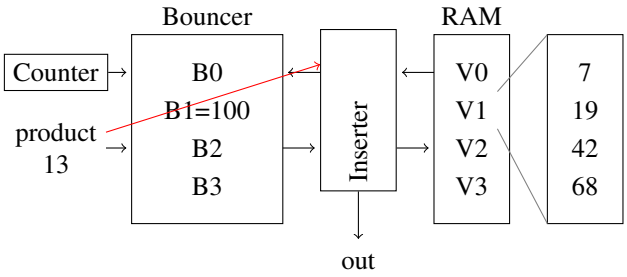
[5] *Virtex-6 Family Overview*, 2nd ed., DS150, Xilinx, Jan. 2012.

[6] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

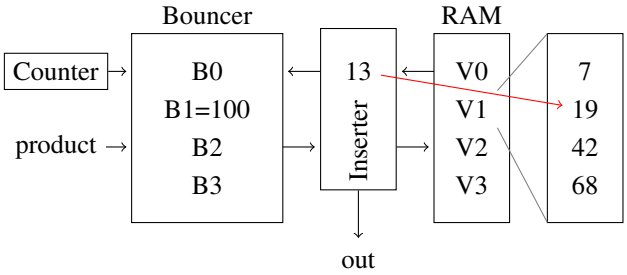
[7] J. S. Beis and D. G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1997, pp. 1000–1006.

[8] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011, pp. 1312–1317.

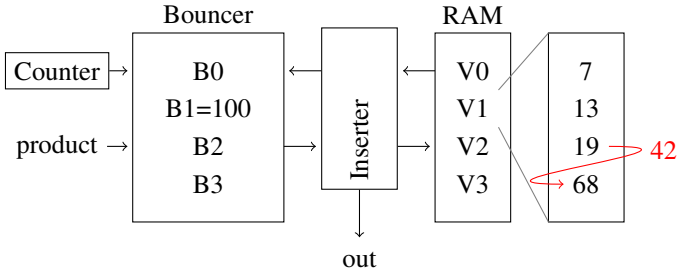
[9] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1998, pp. 604–613.



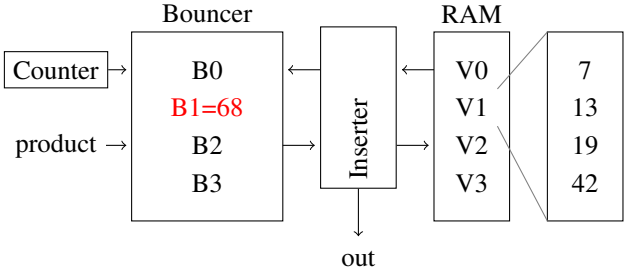
(a) When the value 13 arrives it is compared to the bouncer value. A counter keeps track of the current test vector and therefore the relevant bouncer value. In this case 13 is less than the bouncer value 100 and 13 gets let in to be inserted.



(b) The value 13 gets compared to values starting with the smallest value (7). Once it reaches a larger value (19), the smaller value replaces the larger value.



(c) The values after the inserted value need to shift down one. We use the one clock cycle latency of the RAM to store the overwritten value. This only works with read-first RAM. This latency value then gets written to the next address.



(d) The last value gets sent to the bouncer block to become the new bouncer value. In this way the bouncer value keeps decreasing throughout the k-NN task.

Figure 6: The Sort block keeps track of the nearest neighbors for each test vector. The two main parts of this block are the bouncer that checks to see if a value should be inserted and the inserter that inserts the value into the nearest neighbor array.