Vectorized Hashing for Sketches

Fatih Tasyaran, Kerem Yildirir,Hakan Ogan Alpar,Ali Osman Berk Supervised by Kamer Kaya

Sabanci University, Faculty of Engineering and Natural Sciences

Computer Science and Engineering

Introduction

Searching is one of the most important problems of computer science and it is used extensively in many areas. Hash functions are one-way functions that map an input value to another value. Randomness is essential in hashing because it is not desired to have multiple values resulting in the same hash value. Hashing is also used in data structures where we use the hashed values as indexes of a table and keep the value in the index of a table determined by hash function. This property allows us to do search, insert, delete operations in O(1) time. [1]

1 Intel SIMD Instructions

SIMD is an instruction set available mostly on all current processors. In this project we used AVX (Advanced Vector Extension) and AVX2 instructions which are available for Intel processors since Sandy Bridge architecture. With AVX instructions, it is possible to process 128 bits of data in registers on parallel, with AVX2 this increased to 256 bits, meaning that we can do simple arithmetic and logical operations of 8 32-bit integers at the same time.[2]

2 Hashing Strategies

As our hash functions, we used Multiply-Shift Hash, MurMurHash3 and Tabular Hash. Pseudocodes of these functions can be found in Appendix B of our report. For each hash function, we have developed our work under 2 models; Model 1, one data multiple hash, where we generate multiple hash values from a single data sample, and Model 2, one data one hash, where we generate only one hash value using the same random seed for a single data sample. For both models we implemented the hash functions with SIMD instructions in simple arithmetic and logical operations. This way we were able to process 8 32-bit integers in parallel.

2.1 Model 1

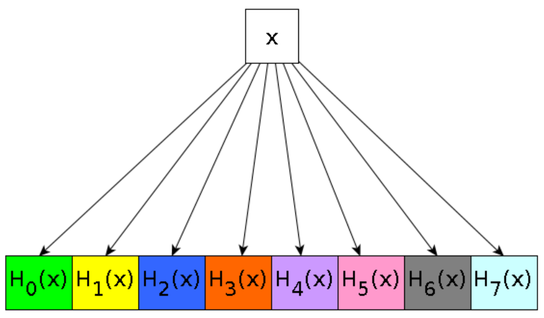


Figure 1: Illustration of Model 1

First model computes 8 hash values for a single given input. We apply the hash function using 8 different random seeds and do all the operations using these random 8 values. We fill an array of size 8 with our input value and apply the operation with 8 different random seeds. This model is especially useful for algorithms like Count-Min Sketch where we keep multiple hash values for a single element.

2.2 Model 2

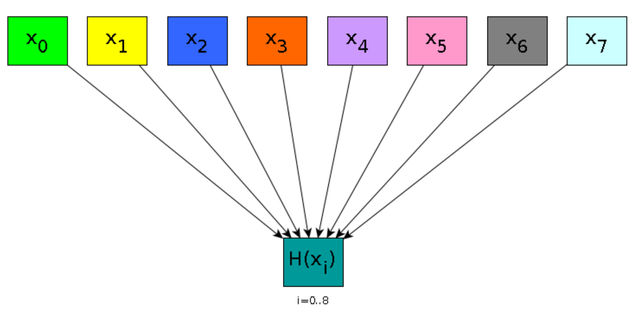


Figure 2: Illustration of Model 2

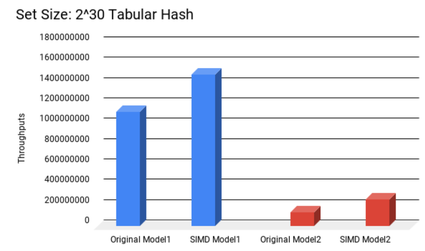
Second model computes 8 hash values for given 8 inputs, using the same random seed for all data points. This approach could be seen as overriding the data length a register can hold, which is 256 bits, by carrying 8 different 32-bit value to get 8 different 32-bit hash result by single instruction, in parallel. Using this approach, the goal is to achieve 8 hash values in a time of 1 hash value without SIMD instructions. This approach could be useful in applications like bloom filter, where every element needs one hash to check membership to a set.

3 Tabular Hash SIMD

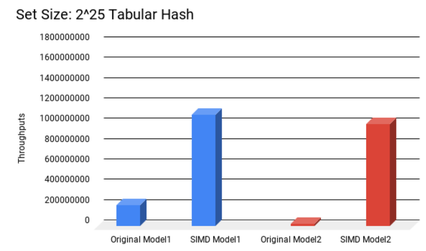
Tabulation hashing is a simple but relatively powerful hash function due to its randomness and efficiency [3]. The algorithm is the following; Let p be the key to be hashed, m be the desired bit length of hash output. M is a k * n matrix and used as a lookup table. Matrix has m bit random elements in every cell.
This algorithm returns a single hash value corresponding to a given single p (key). It is possible to construct two models which are able to return multiple hash values $f_i(p)(i = 0, 1,)$ corresponding to a given a single p and return a multiple hash values as a result of same hash function $f_0(pi)(i = 0, 1, 2, 3.)$ corresponding to given different ps (as a vector) by using SIMD instructions and vectors. The first model takes a single p value as key, but uses a lookup table which has a different type. The algorithm returns a vector of m bit hash values. Initialization of M is the difference between this algorithm and the original one. M contains vectors of random m bits as elements. The speedup is a result of parallel XOR operations.
The other model uses same type of lookup table M But XOR and shifting operations are done by using SIMD nstructions. Algorithm takes a vector of p values as different keys. The result is again a vector of different m bit hash values.

function SIMPLE TABULATION HASH(p,m,M)
  Given p,m, and initialized M(k*n)
  hashvalue=0
  for every i th most significant q bit of p, call it x **do**
    hashvalue = hashvalue $\oplus$ M[i][x]
  return hashvalue
function TABULAR HASH SIMD(p,m,M(k * n))
  Given p, m, M(k * n)
  hashvalue= [0,0,..0]
  for every ith most significant q bit of p, call it x **do**
    hashvalue = _mm256_xor_si256(hashvalue, M[i][x])
  return hashvalue

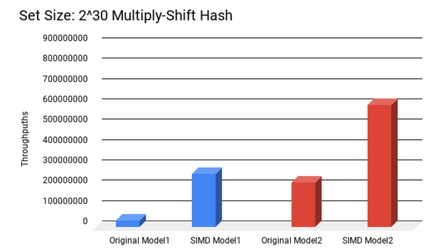
4 Experiment and Results



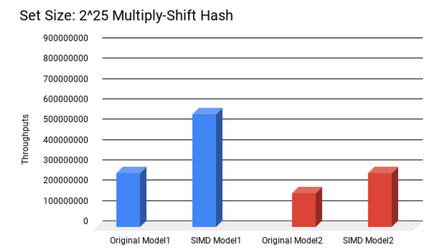
(a) Tabular Hash with 2^{30} elements



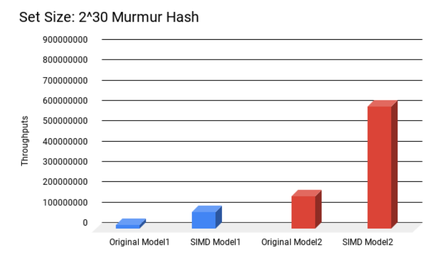
(b) Tabular Hash with 2^{25} elements



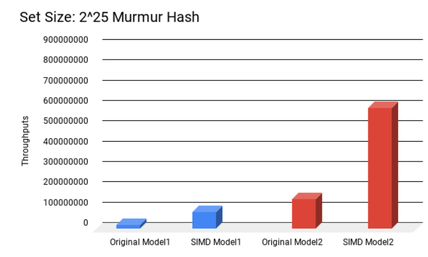
(c) Multiply Shift Hash with 2^{30} elements



(d) Multiply Shift Hash with 2^{25} elements



(e) Murmur Hash with 2^{30} elements



(f) Murmur Hash with 2^{25} elements

Above figures demonstrate the throughputs achieved using SIMD instructions in each hashing function. Main speed-up is due to shift and xor operations with SIMD instructions. Meaning Single Instruction Multiple Data streams, SIMD instructions provide data level parallelism by using single instruction to process multiple data points in parallel. This omits individual fetch ,decode, execute cycles of data points and merges them to one. This approach greatly improves process time in a data-intensive process. Such as hashing a big set of values. As could be seen in the results, using SIMD instructions provides 3 to 6 times speed-up except Tabular Hash. Which accesses a table with randomly filled integers randomly. The overhead with 2^{30} elements caused by this randomness, since the number of elements are growing, accessing the table costs more time.

5 Future Work

In the future, we plan to use our hash functions with SIMD instructions in probabilistic data structures and algorithms such as HyperLogLog, Bloom Filter and Count-Min Sketch. These algorithms are very useful in estimating frequencies in large data streams. We believe that parallelizing the hashing operations in these algorithms will provide great amount of speedup in frequency estimation.

5.1 HyperLogLog

HyperLogLog hashes every key to a bit stream and then approximates the distinct element number of the hashed set by bit stream's prefixes. To do this, HyperLogLog uses the same hash function for all keys to be hashed. With this property, HyperLogLog is suitable for multiple data, one hash approach. Implementing SIMD instructions on a HyperLogLog could achieve great speedup as the HyperLogLog works in a one pass over data manner, only counting hash values.

5.2 Bloom Filter

As mentioned earlier, bloom filter is one of the well suited applications for multiple data - one hash approach. To open up, bloom filter is a membership query which hashes keys to hash values and map them in a bit vector. With a bloom filter employing a SIMD instruction including hash function, it would be possible to answer 8 membership queries at once.

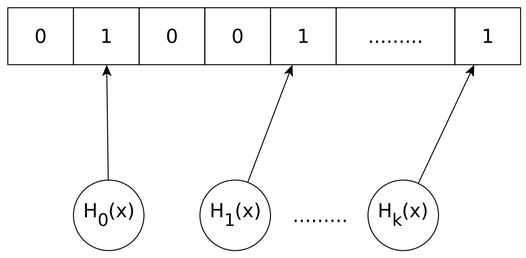


Figure 3: Illustration of an insertion of the element x to a bloom filter with k hash functions

5.3 Count-Min Sketch

Count-min sketch is a perfectly well suited application to use with one data multiple hash approach. Since a count-min sketch is basically a 2 dimensional matrix, the rows of which are 1 dimensional hash tables. To insert an element to a sketch (i.e. counting it) requires different hash values for each row of the count-min sketch. By using SIMD instructions, calculating hash values for all rows a sketch at a time could achieve a promising speed-up for the hashing phase of a count-min sketch.

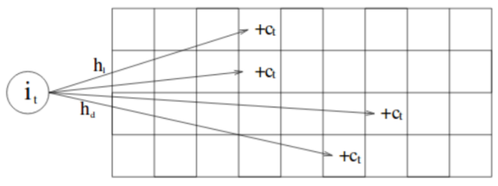


Figure 4: Illustration of Count-Min Sketch

References

[1] S. r. Dahlgaard, M. Knudsen, and M. Thorup, "Practical hash functions for similarity estimation and dimensionality reduction," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 6615–6625, Curran Associates, Inc., 2017.

[2] H. Wang, P. Wu, I. G. Tanase, M. J. Serrano, and J. E. Moreira, "Simple, portable and fast simd intrinsic programming: Generic simd library," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, (New York, NY, USA), pp. 9–16, ACM, 2014.

[3] M. Patrascu and M. Thorup, "The power of simple tabulation hashing," *CoRR*, vol. abs/1011.5200, 2010.