Parallel Hashing

Program for Undergraduate Research

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

January 2019

Contents

1 Introduction

Searching is one of the most important problems of computer science and it is used extensively in many areas such as cryptography and network security. 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. In this work, we introduce using SIMD instructions by Intel in hash functions to generate hash values in parallel by using different data points in a single instruction. Section 2 introduces the SIMD instructions, section 3 discusses the two different hashing approaches we worked on , section 4 presents the improvement we achieved and section 5 presents applications for our work.

2 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. The SIMD instructions we used on this project and their descriptions could be found in Appendix A.

3 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. 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.

3.1 Model 1

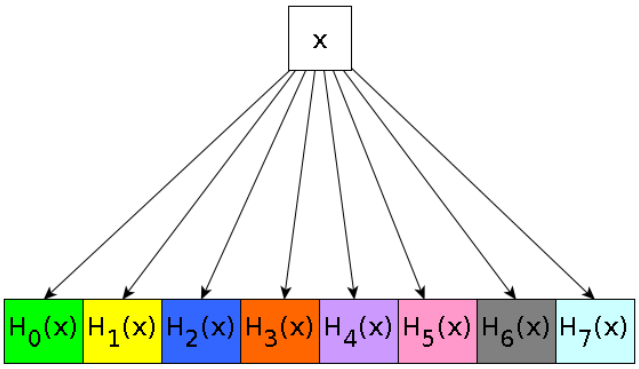

Figure 1: Illustration of Model 1

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.

3.2 Model 2

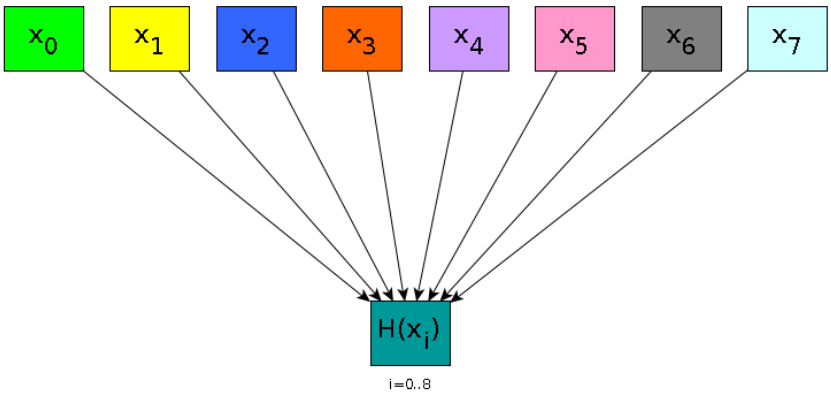

Figure 2: Illustration of Model 2

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.

4 Experiment and Results

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.

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 hashtables. 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, calculatıng 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.

Appendix A SIMD Instructions Used in Project

A.1 Load-Extract Instructions

__m256i _mm256_load_si256 (__m256i const * mem_addr)

Load 256-bits of integer data from memory into dst. mem_addr must be aligned on a 32-byte boundary.

__m256i _mm256_loadu_si256 (__m256i const * mem_addr)

Load 256-bits of integer data from memory into dst.mem_addr does not need to be aligned on any particular boundary.

__int32 _mm256_extract_epi32 (__m256i a, const int b)

Add 4 packed to 256-bit side by side 64-bit integers in a and b, and store the results in dst.

__m256i _mm256_set1_epi32 (int a)

Broadcast 32-bit integer a to all elements of dst.

A.2 Bitwise Instructions

__m256i _mm256_slli_epi64 (__m256i a, int imm8)

Shift packed 64-bit integers in a left by imm8 while shifting in zeros, and store the result in dst.

__m256i _mm256_slri_epi64 (__m256i a, int imm8)

Shift packed 64-bit integers in a right by imm8 while shifting in zeros, and store the result in dst.

__m256i _mm256_shuffle_epi32 (__m256i a, int imm8)

Shuffle 32-bit integers in a within 128-bit lanes using the control in imm8, and store the results in dst.

A.3 Arithmetic Instructions

__m256i _mm256_add_epi32 (__m256i a, __m256i b)

Add 8 packed to 256-bit side by side 32-bit integers in a and b, and store the results in dst.

__m256i _mm256_add_epi64 (__m256i a, __m256i b)

Add 4 packed to 256-bit side by side 64-bit integers in a and b, and store the results in dst.

__m256i _mm256_sub_epi64 (__m256i a, __m256i b)

Subtract packed 64-bit integers in b from 64-bit integers in a, and store the result in dst.

__m256i _mm256_mullo_epi32 (__m256i a, __m256i b)

Multiply the packed 32-bit integers in a and b, producing intermediate 64-bit integers, and store the low 32 bits of the intermediate integers in dst.

A.4 Logical Instructions

__m256i _mm256_and_si32 (__m256i a, __m256i b)

Compute the bitwise AND of 256 bits integer represented data in a and b and store the result in dst.

__m256i _mm256_xor_si256 (__m256i a, __m256i b)

Compute the bitwise XOR of 256 bits integer represented data in a and b and store the result in dst.

Appendix B Hash Functions Used in Project

B.1 Multiply-Shift Hash

```
1: function MULTIPLY-HASH(x)
2:     2 randomly sampled 64 bit integers m_a,m_b ;
       return (m_a * (uint_64) * x + m_b) >> 32
```

B.2 MurMurHash3

```
1: function MURMUR3(key, len, seed)
2:     c1 = 0xcc9e2d51
3:     c2 = 0x1b873593
4:     r1 = 15
5:     r2 = 13
6:     m = 5
7:     n = 0xe6546b64
8:     hash = seed
9:     for each fourByteChunk of key do
10:        k ← fourByteChunk
11:        k = k * c1
12:        k = (k ROL r1)
13:        k = k * c2
14:        hash = hash XOR k
15:        hash = (hash ROL r2)
16:        hash = hash * m + n
                                   ▷ Endian swapping is only necessary on big-endian machines.
17:    for each remaining byte in key do
18:        remainingBytes = SwapEndianOrderOf(remainingBytesInKey)
19:        remainingBytes = remainingBytes * c1
20:        remainingBytes = (remainingBytes ROL r1)
21:        remainingBytes = remainingBytes * c2
22:        hash = hash XOR remainingBytes
23:    hash = hash XOR len
24:    hash = hash XOR (hash SHR 16)
25:    hash = hash * 0x85ebca6b
26:    hash = hash XOR (hash SRH 13)
27:    hash = hash * 0xc2b2ae35
28:    hash = hash XOR (hash SHR 16)
```

B.3 Tabular Hash