Parallel Hashing

Program for Undergraduate Research

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

January 2019

Contents

1 Introduction

Search is one of the most important problems of computer science and used extensively in any areas. The purpose of a hash function is to map any key or value to an index, in order to make a search being possible to do in O(1) time. With this, any operation of, accessing, comparing, compressing etc. of datasets become cheaper to execute since we know that if data is in the dataset by one access because we know it's possible index. With the data that generated is growing everyday, hash functions become a powerful tool and parallelising them is a one way to acquire a fast hash function. In this work, we will introduce Intel's SIMD instructions to achieve instruction level parallelism over hash functions.

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. The SIMD instructions we used on this project and their descriptions could be found in the following paragraph.

2.1 Load-Extract Instructions

__m256i 256_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 256_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.

2.2 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.

2.3 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.

2.4 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.

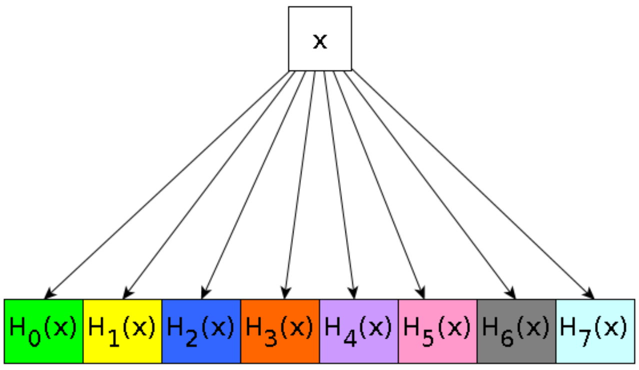
__m256i _mm256_xor_si256 (__m256i a, __m256i b)

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

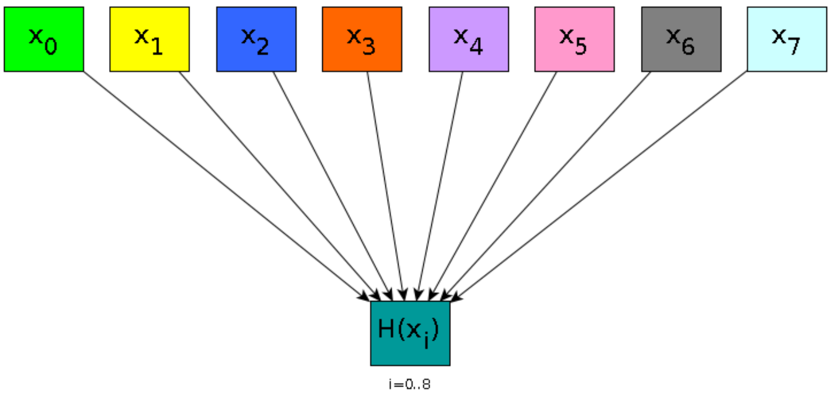
3 Hashing Functions

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. As our hash functions, we used Multiply-Shift Hash, MurMurHash3 and Tabular Hash.

3.1 Model 1



3.2 Model 2



3.3 Multiply-Shift Hash

```
1: function MULTIPLY-HASH(x)
2:     2 randomly sampled 64 bit integers $m_a$,$m_b$ ;
       return $(m_a * (uint\_64) * x + m_b) >> 32$
```

3.4 MurMurHash3

```
function Murmur3(key, len, seed)
    c1 = 0xcc9e2d51
    c2 = 0x1b873593
    r1 = 15
    r2 = 13
    m = 5
    n = 0xe6546b64
    hash = seed
    for each fourByteChunk of key do
        k ← fourByteChunk
        k = k * c1
        k = (k ROL r1)
        k = k * c2
        hash = hash XOR k
        hash = (hash ROL r2)
        hash = hash * m + n
                                ▷ Endian swapping is only necessary on big-endian machines.
    for each remaining byte in key do
        remainingBytes = SwapEndianOrderOf(remainingBytesInKey)
        remainingBytes = remainingBytes * c1
        remainingBytes = (remainingBytes ROL r1)
        remainingBytes = remainingBytes * c2
        hash = hash XOR remainingBytes
    hash = hash XOR len
    hash = hash XOR (hash SHR 16)
    hash = hash * 0x85ebca6b
    hash = hash XOR (hash SRH 13)
    hash = hash * 0xc2b2ae35
    hash = hash XOR (hash SHR 16)
```

3.5 Tabular Hash

4 Experiment and Results

5 Future Work

5.1 HyperLogLog

5.2 Bloom Filter

5.3 Count-Min Sketch

6 Conclusions