



## Parallel Hashing

Program for Undergraduate Research

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

January 2019

## Contents

<b>1</b>	<b>Introduction</b>	<b>i</b>
<b>2</b>	<b>Intel SIMD Instructions</b>	<b>i</b>
2.1	Load-Extract Instructions . . . . .	ii
2.2	Bitwise Instructions . . . . .	ii
2.3	Arithmetic Instructions . . . . .	ii
2.4	Logical Instructions . . . . .	ii
<b>3</b>	<b>Hashing Functions</b>	<b>iii</b>
3.1	Model 1 . . . . .	iii
3.2	Model 2 . . . . .	iii
3.3	Multiply-Shift Hash . . . . .	iii
3.4	MurMurHash3 . . . . .	iv
3.5	Tabular Hash . . . . .	iv
<b>4</b>	<b>Experiment and Results</b>	<b>iv</b>
<b>5</b>	<b>Future Work</b>	<b>iv</b>
5.1	HyperLogLog . . . . .	iv
5.2	Bloom Filter . . . . .	iv
5.3	Count-Min Sketch . . . . .	iv
<b>6</b>	<b>Conclusions</b>	<b>iv</b>

## 1 Introduction

Hash functions are

## 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.

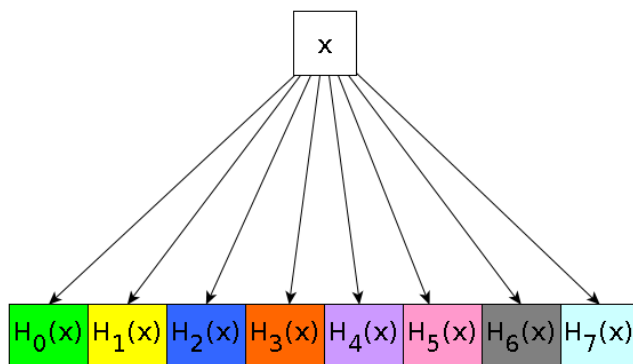
```
__m256i _mm256_xor_si256 ( __m256i a, __m256i b)
```

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

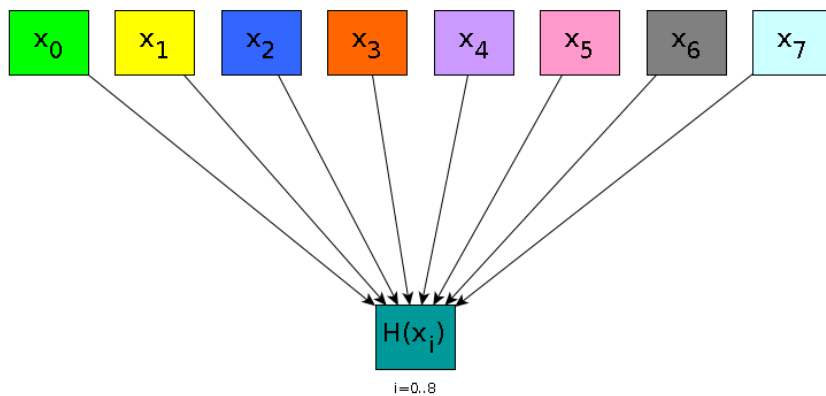
### 3 Hashing Functions

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

---

```

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)

```

---

### 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