Speeding Up Galois Field Arithmetic on Intel MIC Architecture

Kai Feng, Wentao Ma, Wei Huang, Qing Zhang, Yili Gong

Speeding up Galois Field Arithmetic on Intel MIC Architecture*

Kai Feng[1], Wentao Ma[1], Wei Huang[1], Qing Zhang[2], and Yili Gong[1,*]

[1] Computer School, Wuhan University
430072 Hubei, China
[2] Inspur (Beijing) Electronic Information Industry Co., Ltd.
100085 Beijing, China
*yiligong@whu.edu.cn

Abstract. Galois Field arithmetic is the basis of LRC, RS and many other erasure coding approaches. Traditional implementations of Galois Field arithmetic use multiplication tables or discrete logarithms, which limit the speed of its computation. The Intel Many Integrated Core (MIC) Architecture provides 60 cores on chip and very wide 512-bit SIMD instructions, attractive for data intensive applications. This paper demonstrates how to leverage SIMD instructions and shared memory multiprocessing on MIC to perform Galois Field arithmetic. The experiments show that the performance of the computation is significantly enhanced.

Keywords: Galois Field Arithmetic; MIC Architecture; SIMD; Open-MP; Speedup

1 Introduction

From disk arrays [1], cloud platforms [2] to archival systems [3] storage systems must have fault tolerance to protect themselves from data loss. Erasure codes provide the basic technology for the fault tolerance of a storage system. The classic Reed-Solomon code [4] organizes a storage system as a set of linear equations whose arithmetic is Galois Field arithmetic, termed *GF*(2^w). *W* is the length of a word, the basic computing unit. Encoding and decoding of a storage system for fault tolerance are implemented by computing these linear equations by multiplying large regions of bytes by various w-bit constants in *GF*(2^w) and combining the products using bitwise exclusive-or (XOR).

Traditional implementations of Galois Field arithmetic use multiplication tables or discrete logarithms, which limit the speed of its computation. The performance using multiplication is at least four times slower than using XOR [5]. James S. Plank et al. fast Galois Field arithmetic using 128-bit SIMD instruction [6].

* This work is supported by the National Natural Science Foundation of China under Grant No. 61100020.

In late 2012, Intel released its commercial products based on the Many Integrated Core (MIC) architecture [7], targeting to High Performance Computing field for the PetaFLOPS era. It is based on the streamlined x86 core and similar to the architecture of the existing CPUs. Since its architectural compatibility, it can utilize existing parallelization software tools, including OpenMP [8], etc. and specialized versions of Intel's Fortran, C++ and math libraries [9]. Its SIMD instructions are further extended to very wide 512-bit and allow 512-bit numbers to be manipulated on a core simultaneously. MIC's 60 cores also greatly enhance its parallel computing capabilities.

To the best of our knowledge, how to use a computing unit as powerful as a MIC coprocessor for Galois Field arithmetic has not been discussed yet. When the operator size of SIMD instructions extends from 128 bits to 512 bits, though the number of elements keeps at 16, the size of each element changes from 8 bits to 32 bits. With smaller w, e.g. $w = 4$, the spatial utilization ratio is only 1/8 for the multiplication table. The obvious waste needs to be avoided to save memory usage. As to larger w, e.g. $w = 32$, the existed algorithm [6] maps a word into 4 8-bit parts since the element size of 128-bit SIMD instructions is 8-bit, which in-creases complexity and decreases performance. With 32-bit elements, the over-head should be reduced.

This paper will detail how to leverage 512-bit SIMD instructions and shared memory multiprocessing to multiply regions of bytes by constants in *GF(*2^w*)* for $w \in \{4, 8, 16, 32\}$. Each value of w has similar but still different implementation techniques. We will present these techniques and compare the performance of our algorithms on MIC with other approaches on other platforms.

The rest of this paper is organized as follows. The next section describes related work. Section 3 gives description about Erasure Codes and Galois Fields. Section 4 introduces 512-bit instructions used in our algorithms. Section 5 details our algorithms leveraging 512-bit SIMD instructions and OpenMP to multiply regions of bytes by constants in *GF(*2^w*)* for w varying from 4 to 32. Section 6 compares and analyzes the performance of our algorithms and the others. Section 7 is the conclusion and future work.

2 Related Work

Erasure coding is an alternative to replication for fault tolerance as storage systems scale. Traditionally used in the communication field, erasure codes have gained their popularity due to lower spatial requirement under the same reliability.

Many erasure codes are based on Galois Field arithmetic, such as Pyramid codes [10], LRC codes [2], RS codes [11] and F-MSR codes [12], among which the most common one is RS codes. RS codes are used in Bigtable [13] from Google, Cassandra [14] from Facebook and Cleversafe [15]. Microsoft Azure uses LRC codes [2].

Traditional implementations of Galois Field arithmetic adopt multiplication tables or discrete logarithms. There are methods proposed to improve Galois

Field arithmetic, such as Kevin M. Greenan et al. using split multiplication tables and composite fields [16], Jianqiang Luo et al. using bit-grouping tables [17] and H. Peter Anvins approach based on fast multiplication by two [8, 18] and so on.

Recently in [6] James S. Plank et al. present the algorithms of Galois Field arithmetic on CPUs using 128-bit SIMD instructions. As with [6], this paper focuses solely on multiplying regions of bytes by constants. We will exploit 512-bit SIMD instructions as well as OpenMP on MIC coprocessors.

3 Erasure Codes and Galois Fields Arithmetic

Fault tolerance of a storage system is enabled by redundancy. For Galois Field Arithmetic based erasure codes, n disks are partitioned into k disks for original data and m disks for coding information, which is calculated from the original data. When no more than m disks fail, the lost data can be recovered through the remaining disks.

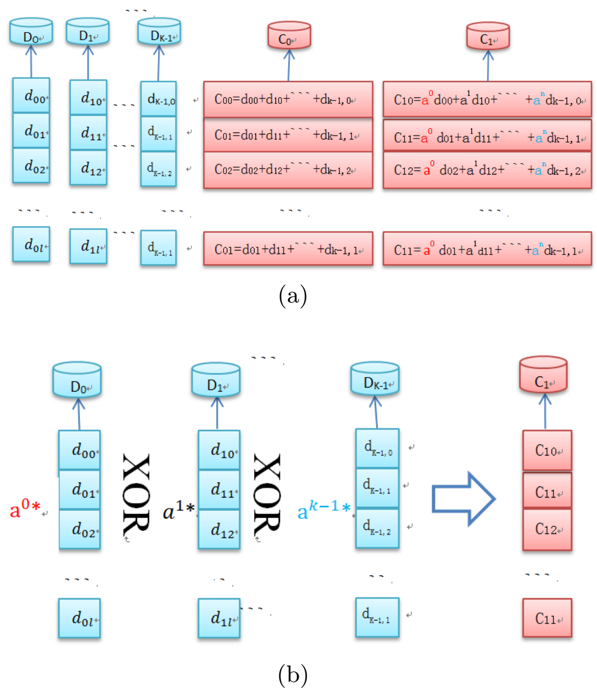


Fig. 1. The RAID-6 data disks and coding disks ($k = 4$). (a) The composition of the RAID-6. (b) How to create code disk C_1.

For example, RAID-6 has two ($m = 2$) coding disks (C_0 and C_1), which are created from k data disks (D_i, $0 \leq i <$n) as shown in Fig. 1 (a). Content of every disk is composed of w-bit words, such as d_{ih} and c_{ih} ($0 \leq i < \text{k}$, $0 \leq j < 2$, $0 \leq h < \text{l}$). Here l is the number of words in a disk. The coding disks are created by a set of linear equations on the right.

The arithmetic of redundant code generation mainly includes Galois Field multiplication and addition, which correspond to multiplication and XOR operations. Taking C_1 as an example, every word d_{ih} is multiplied by a constant a^i, shown in Fig. 1 (b). The products of d_{ih} and a^i ($0 \leq i < \text{k}$) are added (XOR-ed) and the sum is c_{ih} ($0 \leq h < \text{l}$). Since the speed of XOR operations is very fast for modern computes, multiplication becomes the dominant concern with code calculating.

The selection of w decides the number of disks in the storage system for protection. For example, when using Reed-Solomon codes, $w = 4$ means the disk number cannot be larger than 16; $w = 16$ sets the limit to 65,536 disks. The value of w also greatly impacts the computation performance. Larger values of w perform much more slowly than smaller ones. Usually w is a power of 2 to match the size of machine words. Combining all the factors together, typically w is 4 or 8 for storage systems [2, 15] and could be 32 and 64 for security and erasure coding purpose [17].

4 512-bit SIMD Instructions

The Intel Many Core not only has ordinary vector floating-point units, but also uses special registers that enable packed data of up to 512 bits in length for optimal vector graphic streaming SIMD processing. These 512-bit instructions [7] can manipulate sixteen elements of 32 bits or eight elements of 64 bits at a time. In this paper, we use manipulation of 16 elements of 32 bits simultaneously. We leverage the following instructions in our implementations:

- ☐ _mm512_setzero_epi32(void): sets all the elements of the 512-bit vector to zero. Returns a 512-bit vector with all elements set to zero.

- ☐ _mm512_set1_epi32(int a): sets all 16 elements of an int32 result vector to an equal integer value specified by *a*. Returns an int32 vector with 16 elements each equal to integer value specified by *a*.

- ☐ _mm512_slli_epi32(__m512i v2, unsigned int count): performs an element-by-element logical left shift of int32 vector *v2*, shifting by the number of bits given by immediate count. If the shift value specified by this parameter is greater than 31 then the result of the shift is zero.

- ☐ _mm512_srli_epi32(__m512i v2, unsigned int count): performs an element-by-element logical right shift.

- ☐ _mm512_and_epi32(__m512i v2, __m512i v3): performs a bitwise AND operation between int32 vectors *v2* and *v3*.

- ☐ _mm512_xor_epi32(__m512i v2, __m512i v3): performs a bitwise XOR operation between int32 vectors *v2* and *v3*.

- ☐ _mm512_loadunpackhi_epi32(__m512i v1_old, void const* mt): the high 64-byte-aligned portion of the double word stream starting at the element-aligned address *mt* is loaded. It usually works together with the intrinsic _mm512_loadunpacklo_epi32(__m512i v1_old, void const* mt) to load 64 bytes in memory into a 512-bit variable.

- ☐ _mm512_permutevar_epi32(__m512i v2, __m512i v3): this is the real enabling SIMD instruction for *GF(*2^w*)*. It permutes 32-bit blocks of int32 vector *v3* according to indices in the int32 vector *v2*. The ith element of the result is the jth element of *v3*, where j is the ith element of *v2*.

5 Galois Field Arithmetic on MIC

In this section, calculating yA in *GF(*2^4*)*, *GF(*2^8*)*, *GF(*2^{16}*)* and *GF(*2^{32}*)* on MIC are presented respectively.

5.1 Calculating yA in *GF(*2^4*)*

When $w = 4$, each word is composed of four bits, and there are only 16 values that a word may be. All operations are based on a 16 16 multiplication table that is small enough to fit into main memory and can be calculated in advance. A table lookup is needed every four bits, i.e. 2K lookups for a region of 1K bytes.

The SIMD intrinsics operates on operators composed of 16 32-bit elements simultaneously. In the original table, each entry corresponds to the 16 4-bit results of a number y multiplied by 16 numbers from 0 to 15. Storing only 4-bit in a 32-bit element is obviously a waste. Thus we try to merge multiple entries into one in the multiplication table, which is showed in Fig. 2. The products of y and *0x0* to *0xf* from 8 entries are placed in 16 elements from the lowest to highest, and in each element the product from entry *7* on the high end and the one from entry 0 at the low end. Compressing entries 8-15, 16-23 is similar.

Since the processing element of SIMD instructions is 32-bit while $w = 4$, every 32 bits in an element are split into 8 4-bit unit using *mask[i]*, shown in Fig. 3 step (6). Step (7)-(9) calculated *tmp[i]* and should be executed for $0 \leq i < 8$. Finally, perform XOR operation on all tmp values and get yA. Thus 40 SIMD instructions fulfill 128 multiplication operations.

In general the amounts of data to be computed are huge. Dividing data into basic units of 512 bits and there are no data dependence among them. Thus it is natural to parallelize Galois Field Arithmetic by OpenMP exploiting 60 cores on MIC and opens up to 240 threads.

5.2 Calculating yA in *GF(*2^8*)*

When $w = 8$, each word is 8-bit and there are 256 values that a word may have. In principal the method used in *GF(*2^4*)* is applicable to the one in *GF(*2^8*)*. The difference is that the instruction _mm512_permutevar_epi32() only works on 16-element tables (each element is 32-bit), 256 values are too large to fit into a

(1) Original table entry with $y = 0$

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Original table entry with $y = 1$

0000000f	0000000e	0000000d	0000000c	0000000b	0000000a	00000009	00000008
00000007	00000006	00000005	00000004	00000003	00000002	00000001	00000000

……

Original table entry with y = 7

0000000b	0000000c	00000005	00000002	00000004	00000003	0000000a	0000000d
00000006	00000001	00000008	0000000f	00000009	0000000e	00000007	00000000

(2) Compressed table entry with y ranges from 0 to 7

b4692df0	c23d1fe0	58c149d0	2e957bc0	4f1ae5b0	394ed7a0	a3b28190	d5e6b380
618f9e70	17dbac60	8d27fa50	fb73c840	9afc5630	eca86420	76543210	00000000

Fig. 2. Merge eight entries into one in the multiplication table when $w = 4$. Four entries in the original table are merged into one to fit 512-bit registers and variables on MIC. The upper line is high-order 256-bits and the lower line is low-order 256-bits. All variables are presented in hex.

(1) *table* = table entry with $y = 7$

f	e	d	c	b	a	9	8
b4692df0	c23d1fe0	58c149d0	2e957bc0	4f1ae5b0	394ed7a0	a3b28190	d5e6b380
7	6	5	4	3	2	1	0
618f9e70	17dbac60	8d27fa50	fb73c840	9afc5630	eca86420	76543210	00000000

(2) *mask [0] = _mm512_set1_epi32(0xf)*

0000000f	0000000f	0000000f	0000000f	0000000f	0000000f	0000000f	0000000f
0000000f	0000000f	0000000f	0000000f	0000000f	0000000f	0000000f	0000000f

(3) *table = _mm512_srli_epi32(table, 28)*

table = _mm512_and_epi32(table, mask[0])

0000000b	0000000c	00000005	00000002	00000004	00000003	0000000a	0000000d
00000006	00000001	00000008	0000000f	00000009	0000000e	00000007	00000000

(4) *mask[i] = _mm512_slli_epi32(mask[0], i << 2)*   *i*: 1→7

……f…..	……f…..	……f…..	……f…..	……f…..	……f…..	……f…..	……f…..
……f…..	……f…..	……f…..	……f…..	……f…..	……f…..	……f…..	……f…..

(5) *A*

391d9f5a	aaab15c3	63e07c43	fb831623	391d9f6a	aaab15c4	63e07c44	fb831624
391d917b	aaab15c5	63e07e45	fb831525	391d9f8f	aaab19c6	63e07c46	fb831c26

(6) *tmp[i] = _mm512_and_epi32(A, mask[i])* (0⩽*i*<8), here *i* = 2

00000f00	00000500	00000c00	00000600	00000f00	00000500	00000c00	00000600
00000100	00000500	00000e00	00000500	00000f00	00000900	00000c00	00000c00

(7) *tmp[i] = _mm512_srli_epi32(tmp[i], i << 2)* (0⩽*i*<8), here *i* = 2

0000000f	00000005	0000000c	00000006	0000000f	00000005	0000000c	00000006
00000001	00000005	0000000e	00000005	0000000f	00000009	0000000c	0000000c

(8) *tmp[i] = _mm512_permutevar_epi32(tmp[i], table)* (0⩽*i*<8), here *i* = 2

0000000b	00000008	00000002	00000001	0000000b	00000008	00000002	00000001
00000007	00000008	0000000c	00000008	0000000b	0000000a	00000002	00000002

(9) *tmp[i] = _mm512_slli_epi32(tmp[i], i << 2)* (0⩽*i*<8), here *i* = 2

00000b00	00000800	00000200	00000100	00000b00	00000800	00000200	00000100
00000700	00000800	00000c00	00000800	00000b00	00000a00	00000200	00000200

(10) Apply XOR operation *_mm512_xor_epi32()* on *tmp[i]* (0⩽*i*<8) and get yA

9a75ab83	33347829	19c062f9	b4d971e9	9a75ab14	3334783f	19c062ff	b4d971ef
9a75a765	33347838	19c06cf8	b4d978e8	9a75abdb	33347a31	19c062f1	b4d972e1

Fig. 3. Multiplying a 512-bit region A by $y = 7$ in $GF(2^4)$.

16-element variable. Let a be an 8-bit word and a_h and a_l be the high-order 4 bits and low-order 4 bits of a respectively, and we have:

$$a = (a_h \ll 4) \oplus a_l. \tag{1}$$

Thus

$$ya = y(a_h \ll 4) \oplus ya_l. \tag{2}$$

Based on the above analysis, the multiplication table is divided into two, $table_{high}$ which stores the result of $y(a_h{\ll}4)$ and $table_{low}$ which storage the result of ya_l. As with *GF(*2^4*)*, multiplication tables are compressed and occupy 8KB memory. Fig. 4 shows the steps to extract the corresponding content from the compressed lookup tables for _mm512_permutevar_epi32() to permute. Since the lookup content for $y = 7$ is at 24-31 bit of each element in the compressed table entry, both $table_{high}$ and $table_{low}$, it is extracted by right-shifting 24 bits and masked by *0xff*.

(1) *tl* = *table*$_{low}$ entry with *y* = 7

f	e	d	c	b	a	9	8
2d22333c	2a243638	232e3934	24283c30	313a272c	363c2228	3f362d24	38302820
7	6	5	4	3	2	1	0
15121b1c	12141e18	1b1e1114	1c181410	090a0f0c	0e0c0a08	07060504	00000000

th = *table*$_{high}$ entry with *y* = 7

ea1a17e7	9a7a47a7	0adab767	7abae727	37874afa	47e7ea7a	d747ea7a	a727ba3a
4d3daddd	3d5dfd9d	adfd0d5d	dd9d5d1d	90a0f0c0	e0c0a080	70605040	00000000

(2) *mask = _mm512_set1_epi32(0xff)*

000000ff	000000ff	000000ff	000000ff	000000ff	000000ff	000000ff	000000ff
000000ff	000000ff	000000ff	000000ff	000000ff	000000ff	000000ff	000000ff

(3) *tl = _mm512_srli_epi32(tl, 24)*

tl = _mm512_and_epi32(tl, mask)

0000002d	0000002a	00000023	00000024	00000031	00000036	0000003f	00000038
00000015	00000012	0000001b	0000001c	00000009	0000000e	00000007	00000000

(4) *th = _mm512_srli_epi32(th, 24)*

th = _mm512_and_epi32(th, mask)

000000ed	0000009a	0000000a	0000007a	00000037	00000047	000000d7	000000a7
0000004d	0000003d	000000ad	000000dd	00000090	000000e0	00000070	00000000

Fig. 4. Multiplying a 512-bit region A by $y = 7$ in *GF(*2^8*)*.

After acquiring the lookup tables, the remaining steps are similar to the ones with $w = 4$ in Fig. 3, except for step (8) and (9). For $w = 8$, eight 4-bits in an element is indexed by i ($0{\leq}i{<}8$). When i is odd, it means that these 4 bits are high-order of a word; when it is even, these 4 bits are low-order of a word. High-order 4 bits and low-order 4 bits are subject to looking up different tables, $table_{high}$ and $table_{low}$, as well as left-shifting different bits. The revisions are as follows:
(8) for the high-order 4 bits i.e. i is odd
tmp[i] = _mm512_permutevar_epi32(tmp[i], th).
for the low-order 4 bits i.e. i is even
tmp[i] = _mm512_permutevar_epi32(tmp[i], tl).
(9) When i is odd: *tmp[i]* = _mm512_slli_epi32(tmp[i], (i-1) $\ll$ 2).
When i is even: *tmp[i]* = _mm512_slli_epi32(tmp[i], i $\ll$ 2).

5.3 Calculating yA in $GF(2^{16})$

For $GF(2^{16})$ each 16-bit word may have $2^{16} = 64$K values. Since the instruction _mm512_permutevar_epi32() only works on 16-element tables, word a is divided into 4-bit sub-words, named a_3 through a_0:

$$a = (a_3 \ll 12) \oplus (a_2 \ll 8) \oplus (a_1 \ll 4) \oplus a_0. \tag{3}$$

Then

$$ya = y(a_3 \ll 12) \oplus y(a_2 \ll 8) \oplus y(a_1 \ll 4) \oplus ya_0. \tag{4}$$

Thus, we need perform 4 table lookup operations for a 16-bits word. We use compressed tables for data storage. The entries from four tables for a constant y take up 256 bytes and the total memory usage is 8 MB.

5.4 Calculating yA in $GF(2^{32})$

For $w = 32$, the processing is similar. We split each word a (32 bits) into 4-bit sub-words, named a_7 through a_0:

$$\begin{aligned} a = (a_7 \ll 28) \oplus (a_6 \ll 24) \oplus (a_5 \ll 20) \oplus (a_4 \ll 16) \oplus (a_3 \ll 12) \\ \oplus (a_2 \ll 8) \oplus (a_1 \ll 4) \oplus a_0. \end{aligned} \tag{5}$$

Then

$$\begin{aligned} ya = y(a_7 \ll 28) \oplus y(a_6 \ll 24) \oplus y(a_5 \ll 20) \oplus y(a_4 \ll 16) \oplus y(a_3 \ll 12) \\ \oplus y(a_2 \ll 8) \oplus y(a_1 \ll 4) \oplus ya_0. \end{aligned} \tag{6}$$

Thus we need perform 8 table lookup operations for a 32-bit word. Since the element size is 32-bit and the same as the size of Galois Field arithmetic word, w, there is no need for compression. The entries from eight tables for a constant y take up 512 bytes and the total size is 2 TB, which is too large to fit into main memory.

6 Performance Evaluation

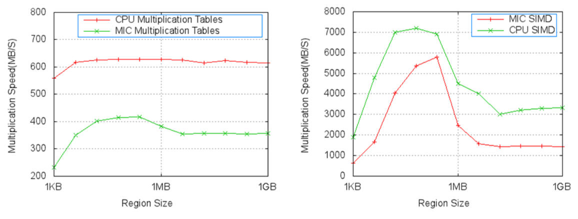
The performance of our proposed algorithms on a MIC coprocessor is evaluated and for comparison the Multiplication Table algorithms [5] and the 128-bit SIMD algorithms from [6] are run on a CPU machine.

The MIC machine used in the experiments is Intel Xeon Phi coprocessor 5110p, 60 cores, core frequency 1.053 GHz, 8 GB GDRR5 memory, 32 KB L1 Instruction Cache, 32 KB L1 Data Cache, 512 KB unified L2 Cache. When the cores do not share data or code, the effective L2 Cache is 30 MB. The comparing machine is Intel Xeon CPU E5620 $*$ 2, 2.4 GHz, 32 KB L1 Instruction Cache, 32 KB L1 Data Cache, 256 KB L2 Cache, 12 MB L3 Cache, 32 GB memory.

The multiplication table algorithms and 128-bit SIMD algorithms are tested on CPU and MIC machines. Our proposed 512-SIMD algorithms are run on MIC with native mode. In all algorithms, regions of random values are multiplied by constants in $GF(2^w)$. For OpenMP accelerated algorithms the region size varies from 1 MB to 1 GB, while for Multiplication Table and SIMD only algorithms the size range is 1 KB to 1 GB. The results are shown in Fig. 5 - Fig. 9.

From Fig. 5 (MulTa is the abbreviation for multiplication table) it can be seen that the SIMD algorithms (128-bit SIMD on CPU and 512-bit SIMD on MIC) greatly outperform the multiplication table algorithms. When $w = 4$, the performance using SIMD on MIC is 13 times more than that of using multiplication table, and 10.6 times on CPU. We can also conclude that the performance of both algorithms on CPU is better than that on MIC, mainly because the core on CPU is more powerful than the one on MIC (2.4 GHz over 1.053 GHz). For example the multiplication table algorithm on CPU is about 1.8 times faster than on MIC and the SIMD is 1.3 times faster. With $w = 8$, 16 and 32 we have similar results and the details are omitted.

Fig. 6 presents the performance under different w values. We can see that the performance does not change much as w grows which is quite different from the conclusion from [6]. In [6] $w = 4$ and $w = 8$ perform roughly the same, $w = 16$ slightly slower and $w = 32$ slower still. This is because MIC SIMD instructions can operate on more bits (512 bits over 128 bits) simultaneously thus fewer operations needed for a word processing, which benefits larger w. For a certain w, when the region size reaches a point between 256 KB and 512 KB, the performance peaks and then drops dramatically. This is because L2 cache saturation impacts the performance greatly.



(a) MulTa algorithms. (b) SIMD algorithms.

Fig. 5. The performance of MulTa algorithms and SIMD algorithms on CPU and MIC with $w = 4$.

The results of OpenMP-based acceleration on the algorithms are shown in Fig. 7 - Fig. 9. For the multiplication table algorithm, it is always CPU-intensive thus changing the region size does little impact on performance as given in Fig. 7. For the 128-bit SIMD algorithms, before L3 cache saturates 8 threads are better than 4 threads; after the saturation they are of the same since it is I/O bound now. In the best case, the 128-bit SIMD outperforms the multiplication table by 9.5.

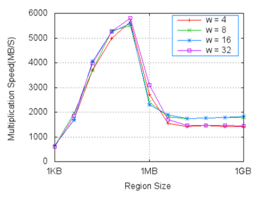


Fig. 6. The performance of SIMD algorithms on MIC with $w = 4, 8, 16$ and 32.

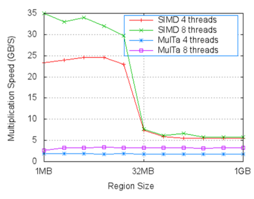


Fig. 7. The performance of OpenMP accelerated MulTa and SIMD algorithms on CPU with 4 and 8 threads, when $w = 4$.

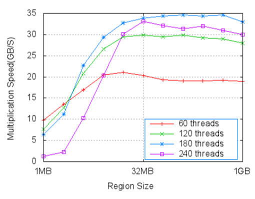
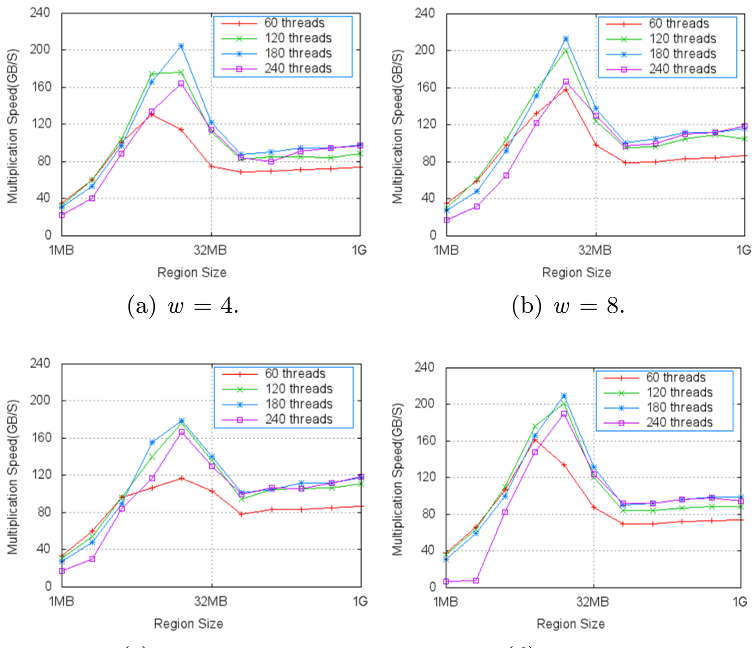


Fig. 8. The performance of OpenMP accelerated MulTa algorithm on MIC, when $w = 4$.



(a) $w = 4$.

(b) $w = 8$.

(c) $w = 16$.

(d) $w = 32$.

Fig. 9. The performance OpenMP accelerated 512-bit SIMD algorithms with different thread numbers on MIC.

Fig. 8 - Fig. 9 compare the performance of the multiplication table algorithm ($w = 4$) with the 512-bit SIMD ($w = 4$, 8, 16 and 32) on MIC. Though each core on MIC is capable of 4-way hardware multi-threading, 240 threads do not have the best performance while generally speaking 180 threads are the best. The 512-bit SIMD + OpenMP algorithm is better than the multiplication table + OpenMP on MIC by 6.8 times and better than the 128-bit SIMD + OpenMP on CPU by 7.2 times.

The peak speedups for all algorithms and conditions are summarized in Table 1 with $w = 4$. Here we take the performance of the single-threaded multiplication table algorithm on CPU as the base 1.

From Fig. 9 (a) - (d) right before the combined 32 MB L2 cache saturates the computing peak can be about 220 GB/s. MIC works as a coprocessor and is connected to the host by standard PCIe x16 which has one-way bandwidth 8 GB/s theoretically. In practice, we have tested that the peak bandwidth from MIC to CPU is 7.0 GB/s and that is 6.7 GB/s from CPU to MIC. Obviously I/O is the bottleneck of Galois Field arithmetic.

Table 1. The speedups with $w = 4$ (taking the performance of the single-threaded multiplication table on CPU as the base 1; MulTa is the abbreviation for multiplication table).

Peak Speedup	MulTa	MulTa+OpenMP	SIMD	SIMD+OpenMP
CPU(128-bit SIMD)	1	5.5	10.6	52.2
MIC(512-bit SIMD)	0.56	55.4	7.2	373.4

7 Conclusion and Future Work

In this paper, we detail how to apply 512-bit SIMD instructions with OpenMP on MIC to Galois Field arithmetic. The algorithms are evaluated with different w from 4 to 32. The performance of our algorithms is about 7.2 to 35.2 times faster than the implementations using 128-bit SIMD with OpenMP on CPU.

With 512-bit SIMD and OpenMP, cache, main memory and I/O to host become bottlenecks. In future we focus on improving the I/O performance and coordination between computation and data transfer.

Acknowledgments. We would like to thank Inspur (Beijing) Electronic Information Industry Co., Ltd offering a server with Intel Xeon Phi coprocessor for experiments.

References

1. Blaum M, Brady J, Bruck J, et al.: EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures[J]. Computers, IEEE Transactions on, 1995, 44(2): 192-202 (1995)

2. Huang C, Simitci H, Xu Y, et al.: Erasure coding in windows azure storage. In USENIX conference on Annual Technical Conference, USENIX ATC (2012)
3. Tansley, R., Bass, M., Smith, M.: DSpace as an open archival information system: Current status and future directions. In Research and advanced technology for digital libraries (pp. 446-460). Springer Berlin Heidelberg (2003)
4. Plank, J. S.: A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software Practice and Experience, 27(9), 995-1012 (1997)
5. J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. WilcoxOHearn.: A performance evaluation and examination of open-source erasure coding libraries for storage. In FAST-2009: 7th Usenix Conference on File and Storage Technologies, pages 253-265 (2009)
6. Plank, J. S., Greenan, K. M., Miller, E. L.: Screaming fast Galois Field arithmetic using Intel SIMD instructions. In FAST-2013: 11th Usenix Conference on File and Storage Technologies, San Jose (2013)
7. Intel Corporation. Intel? C++ Compiler XE 13.1 User and Reference Guides, `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm`
8. OpenMP Application Program Interface, `http://openmp.org/wp/`
9. Intel ® Math Kernel Library for Linux* OS User's Guide, `http://software.intel.com`
10. C. Huang, M. Chen, and J. Li.: Pyramid codes: Flexible schemes to trade space for access efficiently in reliable data storage systems. In NCA07: 6th IEEE International Symposium on Network Computing Applications, Cambridge, MA (2007)
11. Kalcher, S., Lindenstruth, V.: Accelerating Galois Field arithmetic for Reed-Solomon erasure codes in storage applications. In Cluster Computing (CLUSTER), 2011 IEEE International Conference on (pp. 290-298). IEEE (2011)
12. Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang.: NCCloud: Applying network cod-ing for the storage repair in a cloud-of-clouds. In FAST-2012: 10th Usenix Conference on File and Storage Technologies, San Jose (2012)
13. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber.: Bigtable: A Dis-tributed Storage System for Structured Data. OSDI'06: Seventh Symposium on Oper-ating System Design and Implementation, Seattle, WA (2006)
14. Avinash Lakshman, Prashant Malik.: Cassandra: A Decentralized Structured S-torage System. ACM SIGOPS Operating Systems Review, 44(2), 35-40 (2010)
15. J. K. Resch and J. S. Plank.: AONTRS: blending security and performance in dispersed storage systems. In FAST-2011: 9th Usenix Conference on File and Storage Technologies, pages 191?202, San Jose (2011)
16. Greenan, K. M., Miller, E. L., Schwarz, T. J.: Optimizing Galois Field arithmetic for diverse processor architectures and applications. In Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on (pp. 1-10). IEEE (2008)
17. Luo, J., Bowers, K. D., Oprea, A., Xu, L.: Efficient software implementations of large finite fields *GF*(2^n) for secure storage applications. ACM Transactions on Storage (TOS), 8(1), 2 (2012)
18. H. P. Anvin. The mathematics of RAID-6, `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`