Accelerating Big Integer Arithmetic Using Intel IFMA Extensions

Shay Gueron
Department of Mathematics
University of Haifa
Haifa, Israel
e-mail: shay@math.haifa.ac.il

Intel Corporation
Israel Development Center
Haifa, Israel

Vlad Krasnov
CloudFlare, Inc.
San Francisco, USA
vlad@cloudflare.com

Abstract— **Intel has recently announced a new set of processor instructions, dubbed AVX512IFMA, that carry out Integer Fused Multiply Accumulate operations. These instructions operate on *512*-bit registers and compute eight independent *52*-bit unsigned integer multiplications, to generate eight *104*-bit products, and accumulate their low/high halves into *64*-bit containers. Using these instructions requires that inputs are converted to (redundant form) radix 2^{52}, and outputs are converted to the desired representation. This paper demonstrates several techniques for leveraging the AVX512IFMA instructions in order to speed up big-integer multiplications. Although processors that support AVX512IFMA are not yet available at the time this paper is written, we show how currently available public tools can be used for estimating their potential performance benefits. For example, based on these tools, we expect a *2x* speedup for *1024*-bit integer multiplication, over the best currently available method.**

Keywords-big integer arithmetic; processor instructions; SIMD; AVX; AVX512; IFMA

I. INTRODUCTION

Arbitrary-precision arithmetic (aka big-number arithmetic) is an important class of computations, used for various scientific applications where the standard precision is insufficient, and in various public key cryptosystems. This makes compute intensive arithmetic operations on big integers (multiplication in particular) a target for software optimization.

This paper shows how arbitrary-precision multiplications can gain significant performance by "vectorized" implementations that leverage the emerging new AVX512IFMA processor instructions [1].

A. SIMD architectures

SIMD (aka vector architecture), an acronym for "Single Instruction Multiple Data", is a type of architecture that has instructions to carry out, simultaneously, some computation over several independent operands. The operands are called "elements", and reside in special registers that hold a few of them together.

Early SIMD architectures used the MMX instructions (operating on *64*-bit registers), which was followed by the SSE architecture that introduced *128*-bit registers (called XMM's). The Advanced Vector Extensions (AVX) extended SSE in several respects, e.g., introducing non-destructive destination, and floating point operations over *256*-bit registers (called YMMS's). YMM registers are capable of holding eight *32*-bit elements, or four *64*-bit elements [1]. Subsequently, AVX2 architecture was first introduced in Intel's processors (in Architecture Codename "Haswell") in 2013 [2]. It added new integer instructions that operate on the *256*-bit wide YMM registers. The following generation of SIMD architectures is called AVX512, and is expected to appear in one of the upcoming Intel processors. This architecture has *32 512*-bit registers called ZMM's, and promotes most of the existing AVX2 instructions (floating point and integer) to operate on theses wider registers, introduces many new instructions, and new concept of "mask registers".

AVX512 is not a single group of instructions. It rather has several subsets. The basis for AVX512 is the AVX512F instructions (F for foundation). These are expected to be supported on all CPUs with AVX512 capabilities. Other extension families include AVX512CD, AVX512DQ and others, each extending the basic AVX512F set with additional instructions and capabilities.

New AVX512 extensions, called AVX512IFMA (Integer Fused Multiply Accumulate) have been recently introduced [1]. They include two instructions: VPMADD52LUQ and VPMADD52HUQ. These instructions can multiply eight *52*-bit unsigned integers residing in ZMM registers, produce the low (VPMADD52LUQ) and high (VPMADD52HUQ) halves of the *104*-bit products, and add the results to *64*-bit accumulators (i.e., SIMD elements), placing them in the destination register (see additional details in the Appendix). They are designed to support efficient big integers arithmetic using radix 2^{52}.

The motivation behind the definition of the IFMA (short for AVX512IFMA) instructions is that the associated implementation can naturally re-use any hardware that supports similar floating point operations: double precision

1063-6889/16 $31.00 © 2016 IEEE
DOI 10.1109/ARITH.2016.22

arithmetic operates, by its definition [3], with *106*-bit mantissa, and rounds results according to the IEEE specification. This means that the underlying hardware has the computational capacity to multiply up to (theoretically) *53*-bit integers, although these raw results are not exposed to the architecture during the floating point operations. The IFMA architecture leverages this capability. It implicitly harvests *104* bits of the mantissa before the floating point "normalization" occurs.

II. PRELIMINARIES

In this paper, we deal with big-integer multiplication. In this context, the term big-integer refers to integers that are too large to fit in the computer "word", and hence cannot be processed directly by the native processor instructions. Rather, they need to be viewed as a "vector of digits". We assume here that these numbers have at least *512* bits in their binary representation (i.e., that they can be viewed as integers with at least eight *64*-bit digits in radix 2^{64}).

Although we discuss integers here, we point out that arbitrary precision floating-point arithmetic also uses integer operations (independently on the mantissa and the exponent manipulations).

A. The Schoolbook Multiplication

The straightforward multiplication algorithm is called "schoolbook multiplication". Asymptotically, this algorithm is not the fastest method for arbitrarily large numbers, but it underlies (as a building block) other algorithms such as Karatsuba [4] and Toom-Cook [5]. Therefore, accelerating the schoolbook multiplication can be leveraged for accelerating other multiplication algorithms as well. We give one example. In the Karatsuba algorithm, a single *2N*-bit multiplication is traded with three *N*-bit (or *N+1*) multiplications, supplemented with several addition/subtraction operations. Each of the three "smaller" multiplications can, in turn, be implemented as another Karatsuba multiplication, or a schoolbook multiplication. The optimal choice depends on the sizes of the operands. This can (and in practice is) be done recursively, up to some efficiency-based threshold, where, at that point, schoolbook multiplication kicks in.

Schoolbook multiplication of two operands (say, *A* and *B*) views the operands as an array of "digits". The multi-digit operand *A* is multiplied by each single digit of *B*, and the resulting sub-products are aligned and summed up to produce the full product $A \times B$. Software implementation typically uses the native CPU "word" size for the digits. For example, on a *64*-bit processor, radix 2^{64} is the natural choice: the digits are integers *0* to $2^{64} - 1$, encoded as *64*-bit entities. The basic flow of schoolbook multiplication is illustrated in Fig. 1. Note that a carry needs to be propagated during the computations, thus the use of radix 2^{64} (or another power-of-two) with SIMD instructions requires some (cumbersome) manipulations to handle the flow correctly.

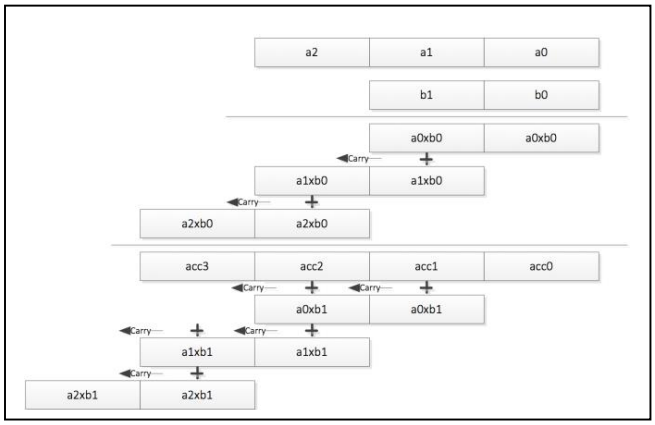


Figure 1. Schoolbook multiplication. An illustration of the carry propagation during the multiplication of the two integers A (3 digits) and B (2 digits). Each digit of A is multiplied by each digit of B, and the appropriate sub-products are aligned and summed up. The digits of the partial sums are not independent of each other, so carries must be propagated from the low to the high digits.

III. THE REDUNDANT REPRESENTATION

To use VPMADD52LUQ and VPMADD52HUQ, we need to convert the operands to a special form that we call "redundant representation". The operands are first written in radix 2^{52}, and the *52*-bit digits are stored in *64*-bit containers that conveniently populate elements of SIMD registers. This representation has *12* bits "leftover" per digit (Fig. 2). It therefore allows VPMADD52LUQ and VPMADD52HUQ to be invoked up to *4096* times, for adding partial products to *64*-bit "accumulators" without generating an overflow. As a result, the cumbersome handling of the carry propagation is avoided. An illustration is shown in Fig. 3.

Let *A* be an *n*-bit integer, written in a radix 2^{64} as an *l*-digits integer, where $l = \lceil n/64 \rceil$. Here, each *64*-bit digit a_i satisfies $0 \le a_i < 2^{64}$, $i = 0, \ldots, l-1$ and $A = \sum_{i=0}^{l-1} a_i \times 2^{64 \times i}$ This representation is unique. Now, consider a positive integer *m* such that $0 < m < 64$. We can write *A* in radix 2^m as $A = \sum_{i=0}^{k-1} x_i \times 2^{m \times i}$. This representation is also unique, and requires $k = \lceil n/m \rceil$; $k \ge l$ digits, x_i, satisfying $0 \le x_i < 2^m$, for $i = 0, \ldots, k-1$.

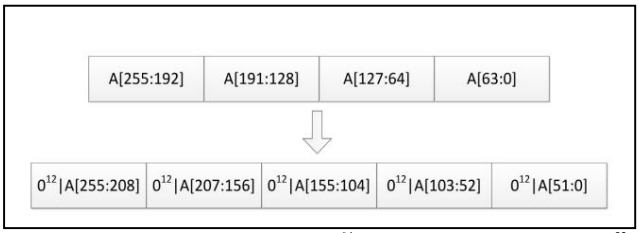
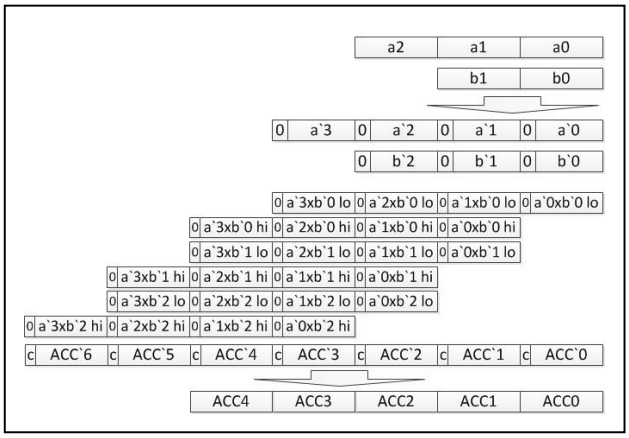


Figure 2. A 256-bit integer in radix 2^{64} (n=256, l=4) written in radix 2^{52} (m=52) using 5 digits (k = ⌈ n/m ⌉ = 5). Each digit can be stored as a 64-bit "element" in a vector of k=5 elements.

Figure 3. Illustration of carry accumulation during a multiplication of two integers, A and B, using IFMA computations. A (3 digits) and B (2 digits)



are first converted into redundant representation in radix 2^{52}. In this representation, they have 4 and 3 digits, respectively, where each digit is smaller than 2^{52}, and is stored in a 64-bit "container" (SIMD element). Then, the sub-products are accumulated, while the carry bits spill into the "empty space" inside the 64-bit container (which is initially set to 0). In the end, the result is converted back to radix 2^{64}.

If we relax the requirement $0 \le x_i < 2^m$, and allow the digits to satisfy only the (looser) inequality $0 \le x_i < 2^{64}$, we say that A is written in a Redundant-radix-2^m Representation (redundant representation for short). This representation is *not* unique.

Informally, the redundant representation can be described as "embedding the digits in a large container".

An integer A, satisfying $A < 2^{m \times k}$, written with k digits in a redundant representation, can be converted to a radix 2^m representation with the *same* number of digits (i.e., k), as shown in Algorithm 1 (Fig. 4). We call this representation "normalized".

Example 1: take $n=1024$ and the 1024-bit number $A = 2^{1024}-105$. It has $l = 16$ digits in radix 2^{64}. The least significant digit is *0xFFFFFFFFFFFFFF97*, and the other *15* digits are *0xFFFFFFFFFFFFFFFF*. With $m = 52$, A becomes a number with $k = 20$ digits in radix 2^{52}. Its least significant digit is *0xFFFFFFFFFFF97*, its most significant digit is *0xFFFFFFFFF*, and the other digits are *0xFFFFFFFFFFFFF*.

```
Algorithm 1: Redundant-to-2^m
Input: U in redundant representation using k
digits (assumption: U < 2^(m×k))
Output: U in radix 2^m representation
Flow:
  1. temp = 0
  2. For i = 0 to k-1
        a. temp = temp + u_i
        b. v_i = temp mod 2^m
        c. temp = temp/2^m
  3. Return V
```

Figure 4. Redundant-to-2^m conversion (see explanation in the text).

Operations on integers that are given in redundant representation can be "vectorized", as follows: let X and Y be two numbers given in redundant representation, such that $X = \sum_{i=0}^{k-1} x_i \times 2^{m \times i}$, $Y = \sum_{i=0}^{k-1} y_i \times 2^{m \times i}$, with $0 \le x_i, y_i < 2^{64}$. Let $t > 0$ be an integer.

Addition: If $x_i + y_i < 2^{64}$ for *i = 0, 1, ..., (k-1)*, then the sum $X + Y$ is given, in redundant representation, by $X + Y = \sum_{i=0}^{k-1} z_i \times 2^{m \times i}$ with $z_i = x_i + y_i; 0 \le z_i < 2^{64}$. If $x_i + y_i \ge 2^{64}$ for some i then normalization to radix 2^{52} is required prior to multiplication.

Multiplication by constant: If $t \times x_i < 2^{64}$ for *i = 0, 1, ..., (k-1)*, i.e. *bitsize(t) + max(bitsize(x_i))<65* then the product tX is given by $tX = \sum_{i=0}^{k-1} z_i \times 2^{m \times i}$ with $z_i = t \times x_i; 0 \le z_i < 2^{64}$.

IV. Vectorizing the Schoolbook Multiplication

Vectorizing the multiplication of two integers (A, B) that are given in a normalized 2^{52} form can be implemented with the IFMA instructions straightforwardly. The flow is described in Algorithm 2 (Fig. 5). Note that the output (X) of Algorithm 2 is given in a non-normalized 2^{52} representation. Several operations can be applied straightforwardly to X as follows.

1. Another big-integer, also given in 2^{52} representation (possibly the output of another multiplication, such as in Karatsuba multiplication) can be added to X.
2. X may be multiplied by a small constant. The bit-size of constant is limited by the number of IFMA instructions that were invoked in order to compute X. If the number of operations is z then the size of the constant is at most $12 - \lceil \log_2(z) \rceil$.
3. X can be converted to normalized 2^{52} representation using Algorithm 1. We call this operation "cleanup".
4. X can be converted to a radix-2^{64} representation.

Operations 1 and 2 can be efficiently vectorized. Operation 1 can be performed with the use of the VPADDQ instruction. Operation 2 can be performed with VPMULUDQ. Obviously, when possible, an application should try to avoid unnecessary conversions of inputs and outputs (operations 3 and 4), in order to save the associated conversion overheads.

Snippets that demonstrate an efficient implementation of operations 3 and 4 are provided in Appendix 2.

V. Implementation and Optimizations

A. Motivation

Currently, the fastest method to implement schoolbook multiplication on a CPU, that we are aware of, uses the classical ALU approach. It uses an ALU instruction to multiply each pair of *64*-bit digits from both operands, individually. The instruction produces two *64*-bit outputs (for the low and high half) which are added to the final result using ALU instructions for addition with carry.

To accelerate this approach, Intel has recently introduced a new set of *64*-bit instructions: MULX, ADCX and ADOX. MULX performs the multiplication and ADCX with ADOX perform the addition with carry. They were first introduced in the Architecture Codename Broadwell.

The classic ADD/ADC instructions set all arithmetic flags according to the result, and carry propagated by setting and consuming the carry flag. This creates dependency which limits the maximal throughput to one ADC operation per cycle. ADCX/ADOX avoid modifying any flags except the carry flag for ADCX and the overflow flag for ADOX they use for carry propagation accordingly. This means that every cycle the CPU can issue one ADCX and one ADOX instruction, resulting in higher throughput when adding large integers.

To justify our vectorized approach, we consider the schoolbook multiplication, and count operations for a) vectorized flow that uses the IFMA instructions; b) vectorized flow uses AVX512 instructions without IFMA; c) ALU flow that uses MULX/ADCX/ADOX instructions.

Computing a multiplication of two *n*-bit numbers in radix 2^m redundant representation requires $[n/m]^2$ scalar multiplications. Similarly, the ALU implementation (in radix 2^{64}) requires $[n/64]^2$ scalar multiplications.

The ALU implementation requires two scalar additions per scalar multiplication (the output of the multiplication are two words). With AVX512 or AVX2, every scalar multiplication is followed by a one single precision addition. By comparison, the IFMA based multiplication requires two operations (to obtain the low and the high halves of the product), but addition can be fused into the calculation (by the definition of the IFMA instructions).

For example, consider the *1024*-bit multiplication. An ALU implementation on a *64*-bit architecture treats the inputs as *16* words, requiring 16^2 and approximately 2×16^2 additions, and the total is *768* operations. Here, each operation requires a single ALU instructions.

An AVX2/AVX512 implementation converts the numbers to radix 2^{29}, each with *36* redundant words. The number of operations required is 36^2 multiplications and 36^2 additions. The total is *2592* operations. However, a single vector instruction is capable of performing *4* operations with AVX2, and *8* operations with AVX512, so the actual number of instructions is *648* and *324*, respectively (this estimation ignores the flow for converting to/from radix 2^{29} to/form 2^{64}).

Radix 2^{29} for AVX2/AVX512 implementation was chosen as it provides the best balance between the number of multiplication operations and the number of required "cleanup" operations.

In our IFMA implementation, the inputs are converted into *20*-digit redundant 2^{52} representations. In total *800* (2×20^2) multiply-accumulate operations are required, which only need *100* AVX512IFMA instructions. Subsequently, the numbers are converted back to radix 2^{64}. The flow is substantially simpler compared to the radix 2^{29} conversion, because we have a more compact representation and can also use the AVX512VBMI instructions.

Algorithm 2: Vectorized implementation of A × B with IFMA
Input: A and B, in radix 2^{52}. Assume A has m digits, and B has n digits.
Output: X = A × B
Conventions:
B_i; X_i; T; Tp – 512 bit wide SIMD registers, with 8, 64-bit elements
$X_i[j]$ – the j^{th} 64-bit element of the 512 bit wide SIMD register X_i
X[i] – X is a pointer to an array of 64 bit values, and i is the addressed integer index
Broadcast – duplicate a single 64 bit value, across 8 element of a 512 bit wide SIMD register
Flow:
1. X_{q-1}, …, X_0 = 0 ($q = \lceil n/8 \rceil$)
2. load digits b_0, b_1, …, b_{1-1} (of B) into SIMD registers B_{q-1}, …, B_0, i.e. B_0 contains b_0 through b_7, B_1 contains b_8 through b_{15} etc.
3. T = 0
4. addCounter = 0
5. for i = 0 to m
   5.1. Tp = T
   5.2. if i = m
        5.2.1. T = 0
   5.3. else
        5.3.1. T = Broadcast digit a_i (of A)
   5.4. for j = 0 to q-1
        5.4.1. $X_j = X_j + low\_52bit(B_j \times T)$
        5.4.2. $X_j = X_j + top\_52bit(B_j \times Tp)$
   5.5. Store X_0[0]at X[i]
   5.6. X_{q-1}, …, $X_0 = X_{q-1}$, …, X_0 >> 64
   5.7. addCounter = addCounter + 2
   5.8. if addCounter ≥ $(2^{64-m})-2$
        5.8.1. perform X "cleanup"
        5.8.2. addCounter = 0
6. Store X_{q-1}, …, X_0 starting at X[m+1]

Figure 5. An algorithm for vectorized multiplication on an IFMA-like supporting architecture, where the multiplication product of two *m*-bit numbers is returned in two *m*-bit halves using two instructions.

B. Implementation

For this study, we wrote code that implements optimized multiplication functions for operand sizes *1024*-bit, *2048*-bit, *3072*-bit and *4096*-bit. For each size, we compared four implementations.

The baseline is the implementation in GMP developer branch [6]. GMP uses schoolbook multiplication for numbers up to *28* digits long (*1792*-bit), and uses the Karatsuba equivalent, Toom-Cook-2 multiplication, for other tested sizes. The developer branch already uses the aforementioned MULX/ADCX/ADOX instructions on

supporting processors, and this is the implementation we test.

In addition, we wrote our own implementation of Karatsuba multiplication, dedicated for the given sizes, with highly optimized underlying schoolbook multiplication that uses MULX/ADCX/ADOX for *384* and *512*-bit operands.

For the vectorized implementations, we wrote an AVX512F based implementation and an AVX512IFMA based one. The AVX512F implementation uses redundant representation in radix 2^{29} for the *1024*-bit operands and radix 2^{28} for the other sizes, while the AVX512IFMA uses radix 2^{52}.

VI. Results

At the date of writing this paper, CPUs that support the AVX512IFMA (or AVX512F) instructions are not yet available. Therefore, the metric we can provide is the number of instructions required per a single invocation of the multiplication function. The instructions count does not map directly to CPU cycles, because dependencies on other micro-architectural features in the processor may play a role in the eventual performance. However, the gap in the number of instructions is substantial, and allows us to confidently claim that the vectorized implementations outperform the ALU alternatives.

The results are summarized in Figs. 6-9. In the figures GMP stands for the baseline implementation of the developer branch of the GMP open source library [6], and ALUX stands for our own implementation that utilizes the new MULX/ADCX/ADOX instructions and is optimized for the specific size.

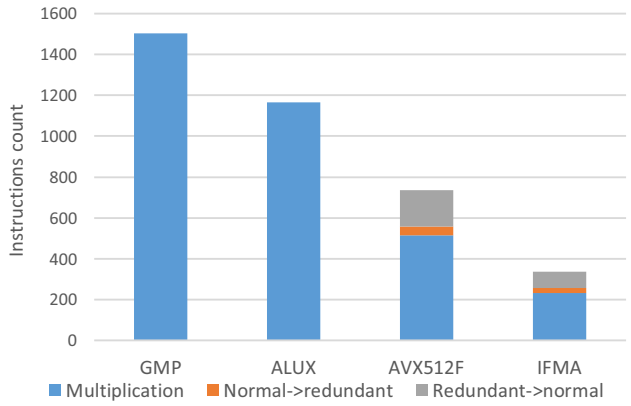


Figure 6. The number of CPU instructions executed per multiplication of two 1024-bit operands. For the vectorized implementations, the number of instructions required for the conversion to and from redundant form is also shown.

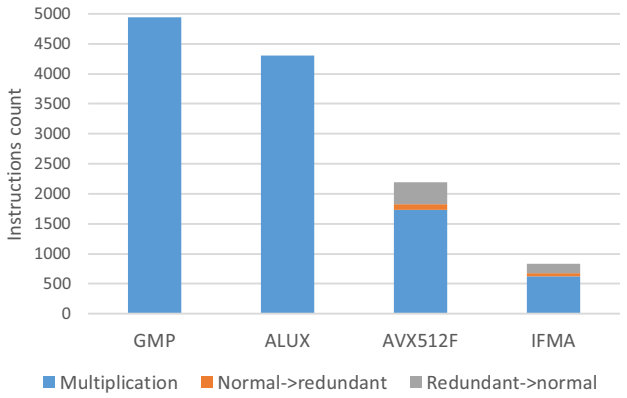


Figure 7. The number of CPU instructions executed per multiplication of two 2048-bit operands. For the vectorized implementations, the number of instructions required for the conversion to and from redundant form is also shown.

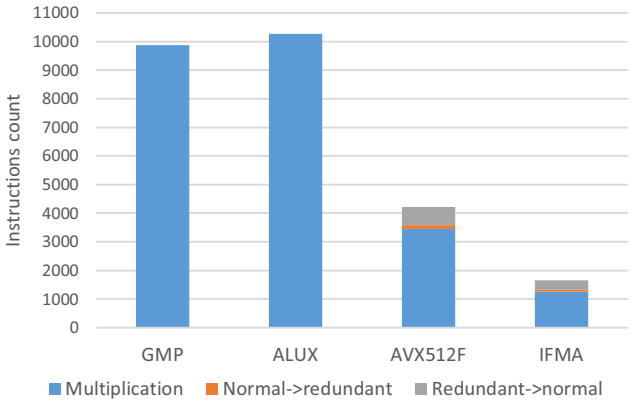


Figure 8. The number of CPU instructions executed per multiplication of two 3072-bit operands. For the vectorized implementations the number of instructions required for the conversion to and from redundant form, is also shown.

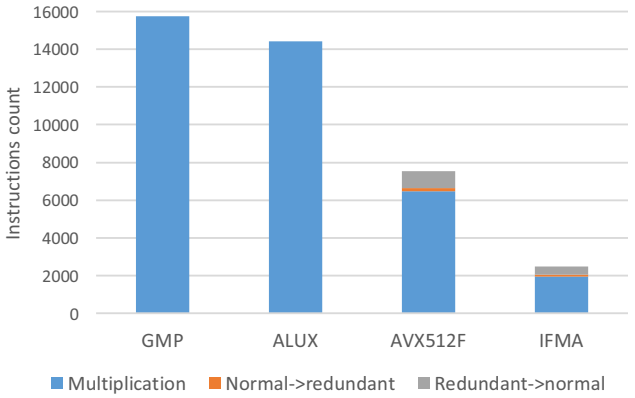


Figure 9. The number of CPU instructions executed per multiplication of two 4096-bit operands. For the vectorized implementations, the number of instructions required for the conversion to and from redundant form, is also specified.

The results clearly show that the vectorized implementations require significantly lower number of instructions. While the gap is smaller for the *1024*-bit multiplication, for the other sizes the IFMA implementation consume less than *20%* of the instructions required by the ALU implementations. The multiplication itself takes less than *15%* of the number of instructions, and the rest is overhead spent on converting the inputs and output to the dedicated form.

VII. Conclusion

The forthcoming IFMA instructions have a great potential to accelerate a class of computations. We introduced here several new implementations for computing large integer multiplication, to greatly accelerate any application that relies on such computations. The projected speedup is hard to estimate accurately at this time, since an actual processor with IFMA is not yet available. However, based on our instructions count methodology, we predict at least 2x speedup compared to the best ALU based implementation for the 1024-bit multiplication, and greater speedups for larger operand sizes.

The implementation can be used as a drop in replacement for large-integer multiplication library. Alternatively, it may serve as a basis for a new library that uses the redundant representation for all the arithmetic operations, thus reducing the conversion overhead for even faster performance. Additional usages of IFMA, e.g., for public key cryptography, are more elaborate, and will be reported in detail, in a future publication.

Our vectorization method is scalable, and can gain performance from any wide (and wider) SIMD architecture.

Finally, we mention that the IFMA based multiplication code, which we reported here, is contributed to the open source community, and is available at [7].

References

[1] Intel Corportaion, “Intel® Architecture Instruction Set Extensions Programming Reference,” Intel Corporation, August 2015. https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

[2] M. Buxton, “Haswell New Instruction Descriptions Now Available!” Intel Corporation, 2011. http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/

[3] IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008, 2008.

[4] A. Karatsuba, and Yu. Ofman, "Multiplication of Multidigit Numbers on Automata," Soviet Physics - Doklady, vol. 7, no. 7, pp. 595 -596, 1963.

[5] A. L. Toom. "The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers," *Soviet Mathematics*, vol. 3, pp. 714 -716, 1963.

[6] GMP, Mercurial repository, accecced September 2015. https://gmplib.org/repo/gmp/

[7] S. Gueron, V. Krasnov, “VPMADD multiplication code,” Github repository, https://github.com/vkrasnov/vpmadd

Appendix 1

To better clarify the way that the AVX512IFMA instructions work, consider the following example:

VPMADD52LUQ DST, SRC1, SRC2

Here, registers DST, SRC1 and SRC2 accommodate (each one) eight 64-bit (SIMD) elements. Each element of SRC1 and SRC2 is populated by the 52 bits of an unsigned 52-bit integer.

The instruction operation performed by “VPMADD52LUQ DST, SRC1, SRC2” is described in Fig. 10.

```
VPMADD52LUQ DST, SRC1, SRC2

FOR j = 0 to 7
       i = j × 64
       tmp[103:0] = SRC1[i+51:i] × SRC2[i+51:i]
       DST[i+63:i] = DST[i+63:i] + tmp[51:0]
ENDFOR
```

Figure 10. Operation of the VPMADD52LUQ instruction.

The VPMADD52HLUQ instruction operates similarly, but adds the top 52-bit part. The addition is integer addition modulo 2^{64}.

As with other AVX512 instructions, AVX512IFMA instructions also support the use of special mask registers, for write and zero masking, for details please refer to [1].

Appendix 2

```
__m512i norm2red(uint8_t in[52]) {
    const __m512i permMask =
        _mm512_setr_epi64(0x0003000200010000,
                          0x0006000500040003,
                          0x0009000800070006,
                          0x000c000b000a0009,
                          0x0010000f000e000d,
                          0x0013001200110010,
                          0x0016001500140013,
                          0x0019001800170016);

    const __m512i shiftMask =
        _mm512_setr_epi64(0,4,8,12,0,4,8,12);

    __m512i data =
        _mm512_maskz_loadu_epi8(0xFFFFFFFFFFFFF,
                                in);
    data =
       _mm512_permutexvar_epi16(permMask, data);
    // The returned value is not a “clean” radix
    // 2^52 number and can only be used with an
    // IFMA instruction that ignores bits above
    // bit 51
    return _mm512_srlv_epi64(data, shiftMask);
}
```

Figure 11. Code snippet. Converting 52-byte input in standard radix 2^{64} Little-Endian represnetation to 8-digit radix 2^{52} representation, using C intrinsic functions.

```
void sbMul(uint64_t out[16], __m512i a,
           __m512i b) {
    int i;
    const __m512i zero = _mm512_setzero_si512();
    __m512i acc = _mm512_setzero_si512();
    __m512i shuf = _mm512_setzero_si512();

    __m512i Bi =
               _mm512_permutexvar_epi64(shuf, b);
    acc = _mm512_madd52lo_epu64(acc, a, Bi);
    _mm512_mask_storeu_epi64(&out[0], 1, acc);
    acc = _mm512_alignr_epi64(zero, acc, 1);

    for (i=1; i<8; i++) {
        acc = _mm512_madd52hi_epu64(acc, a, Bi);
        shuf = _mm512_set1_epi64(i);
        Bi = _mm512_permutexvar_epi64(shuf, b);
        acc = _mm512_madd52lo_epu64(acc, a, Bi);
        _mm512_mask_storeu_epi64(&out[i], 1,
                                 acc);
        acc = _mm512_alignr_epi64(zero, acc, 1);
    }
    acc = _mm512_madd52hi_epu64(acc, a, Bi);
    _mm512_storeu_si512(&out[8], acc);
}
```

Figure 12. Code snippet. Multiplying two 52-byte numbers in redundant representation with AVX512IFMA instructions, using C intrinsic functions.

```
void red2norm(uint64_t out[13], uint64_t in[16]) {
    const __m512i permMask0 =
          _mm512_setr_epi64(0x0706050403020100,
                            0x0f0e0d0c0b0a0908,
                            0x1716151413121110,
                            0x1f1e1d1c1b1a1918,
                            0x30ffffffffff2726,
                            0x3736353433323130,
                            0x3f3e3d3c3b3a3938,
                           0x4746454443424140);
    const __m512i permMask1 =
          _mm512_setr_epi64(0x0f0e0d0c0b0a0908,
                            0x1716151413121110,
                            0x1f1e1d1c1b1a1918,
                            0x2726252423222120,
                            0x2f2e2d2c2b2a2928,
                            0x3a3938ffffffff2f,
                            0x4746454443424140,
                           0x4f4e4d4c4b4a4948);
    const __m512i permMask2 =
          _mm512_setr_epi64(0x18ffffffff0f0e0d,
                            0x1f1e1d1c1b1a1918,
                            0x2726252423222120,
                            0x2f2e2d2c2b2a2928,
                            0x3736353433323130,
                            0xffffffffffffffff,
                            0xffffffffffffffff,
                           0xffffffffffffffff);
    const __m512i permMask3 =
          _mm512_setr_epi64(0x1716151413121110,
                            0x2120070707070717,
                            0x2f2e2d2c2b2a2928,
                            0x3736353433323130,
                            0x403e3d3c3b3a3938,
                            0xffffffffffffffff,
                            0xffffffffffffffff,
                           0xffffffffffffffff);
    const __m512i shiftMask0 =
         _mm512_setr_epi64(0,12,24,36,0,8,20,32);
    const __m512i shiftMask1 =
        _mm512_setr_epi64(52,40,28,16,4,4,32,20);
    const __m512i shiftMask2 =
        _mm512_setr_epi64(4,4,16,28,40,64,64,64);
    const __m512i shiftMask3 =
        _mm512_setr_epi64(8,0,36,24,12,64,64,64);
    const __m512i mOne = _mm512_set1_epi64(-1);

    __m512i r0 = _mm512_loadu_si512(in);
    __m512i r1 = _mm512_loadu_si512(&in[8]);
    // First permute and align all the elements
    __m512i t0 =
         _mm512_maskz_permutex2var_epi8(
          0xFFFFFF83FFFFFFFF, r0, permMask0, r1);
    __m512i t1 =
         _mm512_maskz_permutex2var_epi8(
          0xFFFFE1FFFFFFFFFF, r0, permMask1, r1);
    __m512i t2 =
         _mm512_maskz_permutexvar_epi8(
               0xFFFFFFFFFFFFFF87, permMask2, r1);
    __m512i t3 =
         _mm512_maskz_permutexvar_epi8(
               0xFFFFFFFFFFFFC1FF, permMask3, r1);

    t0 = _mm512_srlv_epi64(t0, shiftMask0);
    t1 = _mm512_sllv_epi64(t1, shiftMask1);
    t2 = _mm512_srlv_epi64(t2, shiftMask2);
    t3 = _mm512_sllv_epi64(t3, shiftMask3);
    t1 =
       _mm512_mask_srli_epi32(t1, 0x400, t1, 8);
    t2 = _mm512_mask_slli_epi32(t2, 0x2, t2, 8);
    // Perform addition
    t0 = _mm512_add_epi64(t0, t1);
    t2 = _mm512_add_epi64(t2, t3);
    // Propagate the carries from the addition
    // We know carry occurred if the addition
    // result is smaller than either of the
    // operands. It may only propagate further
    // if the following element equals max
    // uint64
    uint16_t m0 =
                _mm512_cmp_epu64_mask(t0, t1, 1);
    uint16_t m1 =
                _mm512_cmp_epu64_mask(t2, t3, 1);
    uint16_t m2 =
               _mm512_cmp_epu64_mask(t0, mOne, 0);
    uint16_t m3 =
               _mm512_cmp_epu64_mask(t2, mOne, 0);

    m0 ^= (m1<<8);
    m2 ^= (m3<<8);
    // m0 indicates carry out
    m0 <<= 1;
    // m2 indicates max uint64
    m0 += m2;
    m0 ^= m2;
    // Add the carry according to the mask
    t0 =
        _mm512_mask_sub_epi64(t0, m0, t0, mOne);
    m0 >>= 8;
    t2 =
        _mm512_mask_sub_epi64(t2, m0, t2, mOne);
    _mm512_storeu_si512(out, t0);
    _mm512_mask_storeu_epi64(&out[8], 0x1f, t2);
}
```

Figure 13. Code snippet. Converting 16-digit number in radix 2^{52} redundant representation to 13-digit normal radix 2^{64} representation, using C intrinsic functions.