M.Tech Computer Science Dissertation

Improved SIMD Implementation of Poly1305

A dissertation submitted in partial fulfillment of the requirements for the M. Tech (Computer Science) degree of the Indian Statistical Institute

By
Sreyosi Bhattacharyya
Roll No- CS1608

Under the supervision of
Prof. Palash Sarkar
Applied Statistics Unit
Indian Statistical Institute, Kolkata

CERTIFICATE

This is to certify that the work described in the dissertation titled 'Improved SIMD Implementation of Poly1305' has been done by Sreyosi Bhattacharyya (Roll no- CS1608) under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this Institute and has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other university for the award of any degree or diploma.

(**Prof. Palash Sarkar**)
Indian Statistical Institute, Kolkata

Date: July 9, 2018

ACKNOWLEDGEMENT

This work would not have been possible without the guidance of my supervisor Prof. Palash Sarkar. I would like to express my gratitude to him for his support, patience and optimism that his has provided me during the course of this work. I have learnt a lot from him.

I would like to thank all my teachers at ISI, Kolkata. They have helped me to gain a better understanding of Computer Science.

My parents have been the strongest support of my life. They have always encouraged me to achieve my goals. I express my deepest gratitude to them.

Abstract

Message Authentication Code is an important cryptographic concept which is used for checking message integrity. The Wegman-Carter construction is important in this field. The polynomial-based hash function, Poly1305 as proposed by Daniel J.Bernstein, is a widely used construction. It can be used to instantiate the hash function as required in Wegman-Carter construction. The vectorization of Poly1305 by Shay Gueron and Martin Goll has shown improvement over the known pre-existing implementations.

The algorithm developed by Shay Gueron and Martin Goll has left some scope of improvement both for 256-bit and 512-bit vectorizations. In 256-bit vectorization, improvement has been achieved for messages each of whose number of 16-byte blocks is not a multiple of 4. In 512-bit vectorization, improvement has been achieved for messages each of whose number of 16-byte blocks is not a multiple of 8. For the said cases the alignment of the input is disturbed repeatedly because 4-decimation Horner and 8-decimation Horner for 256-bit and 512-bit vectorizations respectively have been applied incompletely. Goll and Gueron have used Intel Intrinsics for 256-bit and 512-bit vectorizations of Poly1305. For 256-bit vectorization AVX2 has been used. For 512-bit vectorization AVX512 has been used. We have used 4-decimation Horner and 8-decimation Horner throughout the length of input message for 256-bit and 512-bit vectorizations respectively irrespective of the message length. We have obtained better results both for 256-bit and 512-bit vectorizations. Detailed result analysis is available for 256-bit vectorization. The detailed result analysis of 512-bit vectorization is unavailable due to time constraints.

In this report we have shown how to balance a message whose number of 16-byte blocks is not divisible by 4 so that it becomes suitable for application of 4-decimation Horner throughout its length. Same modifications have been done for application of 8-decimation Horner. We also provide a modified SIMD multiplication algorithm for handling messages where in each case, the number of 16-byte blocks when divided by 4 leaves 1 as remainder. Then we give detailed result analysis for Skylake and Kaby Lake cores using suitable graphs and tables.

List of Figures

List of Tables

Contents

1 Introduction

MAC is required for checking message integrity. If the receiver wants to ensure that the message received by it has not been changed en-route then MAC is required. If sender (Alice) sends a message m to the receiver (Bob) and Bob wants to ensure that the message m' that he has received is same as that has been sent by Alice, Bob needs another piece of information t called a tag. So a secret key k is required. Alice computes the tag t from message m using key k and sends $(m,\ t)$ to Bob. Let us assume that Bob receives m'. Now, Bob uses k and m' to compute the tag t'. If t and t' are equal then Bob accepts m' (i.e., $m = m'$). Otherwise, he rejects m' (i.e., $m \neq m'$).

Universal Hash Functions are used for generating the tag. A well-known MAC construction is by Wegman and Carter. Let N be the nonce space, M be the message space. The general concept is as follows: $(N,\ M) \overset{(k,\tau)}{\longmapsto} F_k\,(N) \oplus Hash_\tau\,(M)$, where k and τ are keys. Here F is a pseudo-random function or pseudo-random permutation. One can use univariate polynomial for $Hash_\tau$. One such polynomial-based hash function is Poly1305 [5] which has wide acceptance.

Importance and Uses of Poly1305

- Google uses Poly1305 for authentication in OpenSSL and NSS since March 2013.
  - In Chrome 31 it deployed a new TLS message suite based on Prof. Dan Bernstein's ChaCha20 and Poly1305 algorithms and it has deployed a standardized variant in Chrome49.
  - In Chrome58 Google has removed the pre-standard variants.
  - Till CHROME 68, which is the latest version available, TLS message suits have not been not changed.
  - Google has deployed a new TLS message suite in Chrome that operates three times faster than AES-GCM on devices that do not have AES hardware acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This reduces latency and saves battery life by cutting down the amount of time spent encrypting and decrypting data.

  These imply the importance and acceptance of Poly1305.
- Poly1305 also saves network bandwidth, since its output is only 16 bytes compared to HMAC-SHA1, which is 20 bytes.
- While the ChaCha20-Poly1305 combination is geared towards 32-bit architectures that do not have AES-NI and PCLMULQDQ (or equivalents), performance is a crucial factor, especially on the server side where multiple TLS connections are processed in parallel. Such servers use high-end processors and it is therefore important to make the performance of the ChaCha20-Poly1305 combination comparable to the highly optimized AES-GCM, if it is to be equally attractive for the servers as well.
  More detailed usage can be found in [4] and [1].

Due to its wide acceptance, efficient software implementation of Poly1305 is important. One such efficient implementation has been done by Goll And Gueron [9]. The 256-bit vectorized implementation of Poly1305 on Intel platforms by Martin Goll and Shay Gueron gives significant speed-up compared to the existing non-vectorized implementations. But this implementation [8],[9] requires repeated fresh packing of operand(s) into 256-bit register(s) for messages where the number of 16-byte blocks is not a multiple of 4 due to the incomplete 4-decimation Horner that has been followed. Our improved SIMD implementation reduces the number of machine cycles using a variant of 4-decimation Horner [7].

Our Contribution

Instead of using an incomplete 4-decimation Horner as in [8], we have used complete 4-decimation irrespective of input message lengths. A message whose number of 16-byte blocks is not a multiple of 4 can be said to be in an 'unbalanced' state. So some kind of balancing is required for application of complete 4-decimation Horner. Same concept applies for 8-decimation Horner.

In order to achieve this notion of balancing we have logically 'prepended' to the message required number of zeros(i.e., added at the beginning of the input message) for achieving a balanced form suitable for application of 4-decimation Horner throughout the entire message length. An obvious alternative is to append the required number of zeros. But we have 'prepended' instead of appending. The reason behind it is [8] uses larger decimations for message-lengths greater than 832 bytes. In such cases, appending zeros will increase the number of machine cycles in the multiplication algorithm as discussed in Section 3.4. Since not much information (apart from throughput and latency) has been revealed by Intel in [2] about the intrinsics to be used and the underlying pipeline's architecture, we have avoided appending zeros to the input message. Similarly, we have 'prepended' required number of zeros to achieve balancing for 8-decimation Horner in case of 512-bit vectorization.

Also, for sparse operands, we have given a compact representation which is suitable for complete 4-decimation Horner. The said modifications have been done for both 256-bit and 512-bit vectorizations.

Detailed results have been obtained and analysed for 256-bit vectorization in Skylake and Kaby Lake processors. Significant speed-up has been observed for messages up to 1 KB size. Beyond that, noticeable speed-up has been observed. The following result is for 256-bit vectorization using AVX2.

- We have obtained maximum 34.46% speed-up and an average of 12.58% speed-up till 1 KB size in Skylake processor and maximum 34.63% speed-up and an average of 14.44% speed-up till 1 KB size for Kaby Lake processor.

Beyond 1KB, noticeable speed-up has been observed for 256-bit vectorization. The following result is for messages having size between 1 KB and 4 KB.

- For message size between 1 KB and 2 KB maximum speed-up of 20.69% with average 6.10% in Skylake core and for Kaby Lake Core maximum speed-up is 16.03% with an average speed-up of 6.53% have been obtained.

- For message size between 2 KB and 3 KB maximum speed-up of 13.79% with average 3.92% in Skylake core and for Kaby Lake Core maximum speed-up is 13.79% with an average speed-up of 4.13% have been obtained.

- For message size between 3 KB and 4 KB maximum speed-up of 10.26% with average 2.90% in Skylake core and for Kaby Lake Core maximum speed-up is 13.75% with an average speed-up of 2.64% have been obtained.

For 512-bit vectorization AVX512 intrinsics and 8-decimation Horner have been used. The performance of our work is better than that in [9]. The detailed analysis of result obtained after modification for 512-bit vectorization is not available due to time constraint. We have given performances of implementations in Skylake and Kaby Lake cores for 256-bit vectorizations. It may be noted that some Kaby Lake cores have been designed for mobile devices and such cores support AVX2. Hence our work is also suitable for mobiles devices using such Kaby Lake cores.

2 Preliminaries

2.1 Message Authentication Code

Let M be a finite message space, K be a finite key space and T be a finite tag space. A MAC system (S,V) is a pair of polynomial time algorithms with the following characteristics:

- S is probabilistic polynomial time algorithm which takes in as inputs the message $m \in M$and a key $k \in K$ to produce an output called tag t, where $t \in T$.
- V is a deterministic polynomial time algorithm which takes in as inputs the same key as S, message m and tag t and outputs either accept or reject.

2.2 Hash Functions and MAC

A keyed hash function is a deterministic algorithm which takes a message and key as inputs and the output is called digest. The message space is much larger than the digest space.

Two Kinds of Probabilities. Let us consider three non-empty finite sets $\mathcal{M}$, $\mathcal{K}$,$\mathcal{T}$. Let H_κ be an indexed family of functions such that for each $\kappa \in \mathcal{K}$, $H_\kappa : \mathcal{M} \to \mathcal{T}$.

- For two distinct $M \in \mathcal{M}$, $M' \in \mathcal{M}$, $\Pr[H_\kappa(M) = H_\kappa(M')]$ is called a collision probability.
- Let $\mathcal{T}$ be an additively written group. For distinct $M \in \mathcal{M}$, $M' \in \mathcal{M}$ and $y \in \mathcal{T}$, the probability $\Pr[H_\tau(M) - H_\kappa(M') = y]$ is called a differential probability.

ϵ-Almost Universal Hash Function. The family $\{H_\kappa\}$ is said to be ϵ-almost universal ϵ-AU if for all distinct $M,\ M' \in \mathcal{M}$ the collision probability for the pair (M, M') is at most ϵ.

ϵ-Almost XOR Universal Hash Function. The family $\{H_\kappa\}$ is said to be ϵ-almost XOR Universal if for all distinct $M,\ M' \in \mathcal{M}$ and any $y \in \mathcal{T}$ the differential probability for the triplet (M, M', y) is at most ϵ.

2.3 Polynomial-based Universal Hash Functions

Universal Hash Functions can be constructed using polynomials modulo a prime. Let us define the following hash function H_κ where $m = (m_1, \ldots, m_n)$ be the input message and each $m_i \in F$, $i \in \{1, \ldots, n\}$ and F is a finite field.
$H_\kappa(m_1, \ldots, m_n) = m_1\kappa^{n-1}\ +\ m_2\kappa^{n-2}\ + \ldots\ +\ m_n$, where $\kappa \in F$
Let $m = (m_1, \ldots, m_n)$ and $m' = (m'_1, \ldots, m'_n)$ be two distinct messages. Now, let us find the collision and differential probability for H_κ.

Collision Probability for H_κ. We need to find $\Pr[H_\kappa(m) = H_\kappa(M')]$ or,
$\Pr[H_\kappa(m) - H_\kappa(m') = 0]$.
$H_\kappa(m) - H_\kappa(m') = 0$
or, $(m_1 - m'_1) \cdot \kappa^{n-1} + \ldots + (m_n - m'_n) = 0$
Since, m and m' are distinct all the coefficients are not zero. As the LHS of the above equation is a polynomial of degree $n-1$, it has at most $n-1$ distinct roots. Hence, $\Pr[H_\kappa(m) = H_\kappa(m')] \leq \frac{n-1}{|F|}$. We see that H_κ is $\frac{n-1}{|F|}$-AU.

Differential Probability for kH_κ. We need to find $\Pr[k\ H_\kappa(m) - k\ H_\kappa(m') = y]$. Note that $k\ H_\kappa - k\ H'_\kappa - y$ is a polynomial of degree n. Hence, this polynomial has at most n distinct roots. So kH_κ is $\frac{n}{|F|}$-AXU.

2.4 Horner's Rule

A polynomial $f(\kappa)$ of degree $\kappa - 1$ may be evaluated at a point τ by $\kappa - 1$ multiplications and $\kappa - 1$ additions using Horner's Rule, where all the coefficients belong to a finite field and for $\kappa\mathbf{Horner}_\tau(m_1, ..., m_n)$, κ multiplications and $\kappa - 1$ additions are required.

$\mathbf{Horner}_\kappa(m_1, ..., m_n) = (((m_1\kappa + m_2)\kappa + ...)\kappa + m_{n-1})\kappa + m_n$. Thus, $Horner_\kappa$ is $\frac{n-1}{|F|} - AU$ and $\kappa Horner_\kappa$ is $\frac{n}{|F|}$-AXU.

2.5 Poly1305- A polynomial based hash function

An input message of length L bits is broken down into blocks of 128-bit length. Let there be ℓ ($\ell = \lceil L/128 \rceil$) such blocks. Each of the blocks is padded first with 1 and then with 0 to make it 130-bit long. If the last block is lesser than 128 bits then at first it is padded with 1 and then with required number of zeros to make it 130-bit long. The resulting block is treated as an unsigned little-endian integer. Let us denote the i^{th} such converted block as c_i, where $i \in \{1, \ldots, \ell\}$.

The computation of the tag is written as follows:

$$\left(\left(\left(c_1 \cdot R^\ell + c_2 \cdot R^{\ell-1} + ... + c_\ell \cdot R\right) \bmod\ 2^{130} - 5\right) + K\right) \bmod\ 2^{128},$$

where R and K are the 2 128-bit divisions of the 256-bit key and ℓ is the number of 16-byte blocks of the message.

2.6 Intel Intrinsics

Intrinsics are assembly-coded functions that allow one to use C function calls and variables in place of assembly instructions. Intrinsics are expanded inline eliminating function call overhead. Providing the same benefit as using inline assembly, intrinsics improve code readability, assist instruction scheduling, and help reduce debugging. Intrinsics provide access to instructions that cannot be generated using the standard constructs of the C and C++ languages. Intel intrinsic instructions are C style functions that provide access to many Intel instructions - including Intel SSE, AVX, AVX-512, and more - without the need to write assembly code.

Intel Intrinsics Background. Advanced Vector Extensions (AVX, also known as Sandy Bridge New Extensions) are extensions to the x86 instruction set architecture for microprocessors from Intel and first supported by Intel with the Sandy Bridge processor. AVX provides new features, new instructions and a new coding scheme. AVX2 expands most integer commands to 256 bits. Advanced Vector Extensions 2 (AVX2), also known as Haswell New Instructions,[5] is an expansion of the AVX instruction set introduced in Intel's Haswell microarchitecture. AVX2 makes the expansion of most vector integer SSE and AVX instructions to 256 bits. In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

Intel CPUs with AVX2 support

- Haswell processor, Q2 2013
- Haswell E processor, Q3 2014
- Broadwell processor, Q4 2014
- Broadwell E processor, Q3 2016
- Skylake processor, Q3 2015
- Kaby Lake processor, Q3 2016(ULV mobile)/ Q1 2017 (desktop/ mobile)
- Skylake-X processor, Q2 2017
- Coffee Lake processor, Q4 2017
- Cannon Lake processor, 2018
- Cascade Lake processor, 2018
- Ice Lake processor, 2018

We have used Skylake and Kaby Lake processors for our implementation. Thus we have been able to give speed-up for modern processors.

Major Intel Intrinsics Used In This Work

- $\_mm256\_mul\_epu32\,(\_\_m256i\ a,\ \_\_m256i\ b)$: Multiply the low unsigned 32-bit integers from each packed 64-bit element in a and b, and store the unsigned 64-bit results in dst.
  Operation:
  ```
  FOR $j := 0\ to\ 3$
      $i := j * 64$
      $dst[i+63:i] := a[i+31:i] * b[i+31:i]$
  END FOR
  $dst[MAX:256] :=$ NULL
  ```

- $\_mm256\_set\_epi32\,(int\ e7, int\ e6, int\ e5, int\ e4, int\ e3, int\ e2, int\ e1, int\ e0)$: Set packed 32-bit integers in dst with the supplied values.
  Operation:
  ```
  $dst[31:0] := e0$
  $dst[63:32] := e1$
  ```

```
$dst[95:64] := e2$
$dst[127:96] := e3$
$dst[159:128] := e4$
$dst[191:160] := e5$
$dst[223:192] := e6$
$dst[255:224] := e7$
$dst[MAX:256] := 0$
```

- $\_mm256\_permute4\times64\_epi64\,(\_\_m256i\ a, const\ int\ mm8)$: Shuffle 64-bit integers in a across lanes using the control in $imm8$, and store the results in dst.
  Operations:

```
$SELECT4(src,\ control)$
{
$CASE(control[1:0])$
$0:\ tmp[63:0]\ := src[63:0]$
$1:\ tmp[63:0]\ := src[127:64]$
$2:\ tmp[63:0]\ := src[191:128]$
$3:\ tmp[63:0]\ := src[255:192]$
$ESAC\ RETURN\ tmp[63:0]$
}
$dst[63:0]\ := SELECT4(a[255:0],\ imm8[1:0])$
$dst[127:64]\ := SELECT4(a[255:0],\ imm8[3:2])$
$dst[191:128]\ := SELECT4(a[255:0],\ imm8[5:4])$
$dst[255:192]\ := SELECT4(a[255:0],\ imm8[7:6])$
$dst[MAX:256]\ := NULL$
```

- $\_mm256\_permutevar8\times32\_epi32\,(\_\_m256i\ a,\ \_m256i\ idx)$: Shuffle 32-bit integers in a across lanes using the corresponding index in idx, and store the results in dst.
  Operation:

```
FOR $j := 0$ to 7
$i\ :=\ j*32$
$id\ :=\ idx[i+2:i]*32\ dst[i+31:i]\ :=\ a[id+31:id]$ END FOR
$dst[MAX:256] :=$ NULL
```

- $\_mm256\_add\_epi64\,(\_\_m256i\ a,\ \_\_m256i\ b)$: Add packed 64-bit integers in a and b, and store the results in dst. Operation:

```
FOR $j := 0$ to 3
$i\ :=\ j*64$
$dst[i+63:i]\ :=\ a[i+63:i]\ +\ b[i+63:i]$
END FOR
$dst[MAX:256] :=$ NULL
```

- $\_mm256\_srli\_epi64\,(\_\_m256i\ a,\ int\ imm8)$: Shift packed 64-bit integers in a right by $imm8$ while shifting in zeros, and store the results in dst. Operation:

```
FOR $j := 0$ to 3
    $i\ :=\ j*64$
   IF $imm8[7:0]\ >\ 63$
      $dst[i+63:i]\ :=\ 0$
   ELSE $dst[i+63:i]$ := $ZeroExtend(a[i+63:i] >> imm8[7:0])$
   FI
END FOR $dst[MAX:256] := 0$
```

- $\_mm256\_and\_si256\,(\_\_m256i\ a,\ \_\_m256i\ b)$: Compute the bitwise AND of 256 bits (representing integer data) in a and b, and store the result in dst. Operation:
  $dst[255:0]\ :=\ (a[255:0]\ AND\ b[255:0])$
  $dst[MAX:256] := 0$

- $\_mm256\_blend\_epi32\,(\_\_m256i\ a,\ \_\_m256i\ b,\ const\ int\ imm8)$: Blend packed 32-bit integers from a and b using control mask $imm8$, and store the results in dst. Operation:

```
FOR $j := 0$ to 7
    $i\ :=\ j * 32$
     IF $imm8[j]$
        $dst[i+31:i] := b[i+31:i]$
     ELSE $dst[i+31:i] := a[i+31:i]$
     FI
END FOR
$dst[MAX:256]$ := NULL
```

- $\_mm256\_slli\_epi64\,(\_\_m256i\ a,\ int\ imm8)$: Shift packed 64-bit integers in a left by $imm8$ while shifting in zeros, and store the results in dst.
  Operation:

```
FOR $j := 0$ to 3
     $i\ :=\ j * 64$
    IF $imm8[7:0] > 63$
      $dst[i+63:i] := 0$
    ELSE
    $dst[i+63:i] := ZeroExtend(a[i+63:i] << imm8[7:0])$
    FI
END FOR
$dst[MAX:256]$ := NULL
```

3 Previous Work: 256-bit Vectorization of Poly1305 by Goll and Gueron

3.1 5-limb representation of an operand and a related operation

Each 130-bit unsigned integer is represented as 5-digit number in base 2^{26}. Let Y be such an integer.

$$Y = y_4 \cdot 2^{4\cdot 26} + y_3 \cdot 2^{3\cdot 26} + y_2 \cdot 2^{2\cdot 26} + y_1 \cdot 2^{26} + y_0 \tag{1}$$

, where each of the 5 limbs is placed in a 32-bit register. Let X be another 130-bit integer represented in the same way. Thus, if we want to multiply (modulo $2^{130}-5$) X and Y, the multiplication is done as follows:

$$z_0 \;=\; y_0 \cdot y_0 + 5 \cdot x_1 \cdot y_4 + 5 \cdot x_2 \cdot y_3 + 5 \cdot x_3 \cdot y_2 + 5 \cdot x_4 \cdot y_1$$

$$
\begin{aligned}
z_1 &= x_0 \cdot y_1 + x_1 \cdot y_0 + 5 \cdot x_2 \cdot y_4 + 5 \cdot x_3 \cdot y_3 + 5 \cdot x_4 \cdot y_2 \\
z_2 &= x_0 \cdot y_2 + x_1 \cdot y_1 + x_2 \cdot y_0 + 5 \cdot x_3 \cdot y_4 + 5 \cdot x_4 \cdot y_3 \\
z_3 &= x_0 \cdot y_3 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_3 \cdot y_0 + 5 \cdot x_4 \cdot y_4 \\
z_4 &= x_0 \cdot y_4 + x_1 \cdot y_3 + x_2 \cdot y_2 + x_3 \cdot y_1 + x_4 \cdot y_0
\end{aligned}
$$

Note: $z_0,\ z_1,\ z_2,\ z_3,\ z_4$ are not in a 5-limb format. In order to represent them in 5-limb format we should do a reduction.

The total number of multiplications is 25 if $(5 \cdot x_1 5 \cdot x_2, 5 \cdot x_3, 5 \cdot x_4)$ or $5 \cdot X$ is precomputed. Since each of the multiplications is 32×32, we need 5 64-bit registers to store them.

3.2 Alignments

Let us denote four blocks of the message as M_0, M_1, M_2 and M_3. Each of them is represented according to the 5-limb representation. So we can denote M_i as $(m_{i0}, m_{i1}, m_{i2}, m_{i3}, m_{i4})$ where $i \in \{0, 1, 2, 3\}$. Let us denote the intermediate results of 32×32 multiplications $(M_i \cdot R^4)$ corresponding to each of the chunks $M_i,\ i \in \{0, 1, 2, 3\}$ as $t_{i0},\ t_{i1},\ t_{i2},\ t_{i3},\ t_{i4}$. Note that R is the 128-bit key which comes as input. If immediately after these multiplications the intermediate results are reduced modulo $2^{130} - 5$, then the reduced intermediate results can be stored in 3 256-bit registers.

Alignment of reduced intermediate results (message in the first step) in 3 256-bit registers is as follows:

t_{32}	t_{30}	t_{22}	t_{20}	t_{12}	t_{10}	t_{02}	t_{00}
t_{33}	t_{31}	t_{23}	t_{21}	t_{13}	t_{11}	t_{03}	t_{01}
x	t_{34}	x	t_{24}	x	t_{14}	x	t_{04}

In the first step 130×4 bits of message is aligned into 3 256-bit registers as shown below.

m_{32}	m_{30}	m_{22}	m_{20}	m_{12}	m_{10}	m_{02}	m_{00}
m_{33}	m_{31}	m_{23}	m_{21}	m_{13}	m_{11}	m_{03}	m_{01}
x	m_{34}	x	m_{24}	x	m_{14}	x	m_{04}

Apart from the intermediate results (or 4 message blocks in first step), the other operands for vectorized multiplication are R^4 or $(g_4, g_3, g_2, g_1, g_0)$ and $5 \cdot R^4$ or $(5 \cdot g_4, 5 \cdot g_3, 5 \cdot g_2, 5 \cdot g_1)$ which are stored in 2 256-bit registers in the following way:

$5 \cdot g_4$	$5 \cdot g_3$	$5 \cdot g_2$	g_4	g_3	g_2	g_1	g_0
$5 \cdot g_1$	$5 \cdot g_1$	$5 \cdot g_1$	$5 \cdot g_1$	$5 \cdot g_1$	$5 \cdot g_1$	$5 \cdot g_1$	$5 \cdot g_1$

3.3 Partitioning the message space

In [9], which has been implemented according to [8] the following message partitioning concept has been used. Since 256-bit vectorization is followed, four 16-byte blocks are handled simultaneously. Let n be the total number of bytes in the input message. Note that $n = \lceil L/8 \rceil$. So if $16|n$ and $4|\ell$ then a 4-decimation Horner [7] is followed for evaluation. Else, if $16|n$ but $4 \not| \ell$, then 4-decimation Horner

[7] is followed up to the length where number of 16-byte blocks is a multiple of 4. The rest of the $n' \, (= n - (\ell * 16))$ bytes are handled using one of the following methods. Let P be the result up to the length for which 4-decimation Horner is possible without appending extra zeros (logically/physically).

- $16|n'$. Let $\ell' = n'/16$
  - case 1: $\ell' = 1$. $P = PR + M_{\ell-1}R$. Fresh packing is required only once.
  - case 2: $\ell' = 2$. $P = PR^2 + M_{\ell-2}R^2 + M_{\ell-1}R$. Fresh packing is required only once.
  - case 3: $\ell' = 3$. $P = PR^3 + M_{\ell-3}R^3 + M_{\ell-2}R^2 + M_{\ell-1}R$. This case is again broken down into 2 cases. First 32 bytes is handled as if ℓ'=2. Then the last 16 bytes or less is processed as if $\ell' = 1$. This requires fresh packing of operands three times.
- $16 \nmid n'$.
  - $1 \leq n' \leq 15$: Case 1 is followed. Fresh packing is required once.
  - $17 \leq n' \leq 31$: Case 1 is followed, where 16-bytes are considered every time till the message ends. Fresh packing is required twice.
  - $33 \leq n' \leq 47$: Uses similar concepts as the previous case. Fresh packing is required twice.
  - $49 \leq n' \leq 63$: For first 32-byte case 2 is followed and for the rest of the bytes case 1 is followed, where 16-bytes are considered every time till the message ends. Fresh packing of operands is required three times.

If $16 \nmid n$, then $\ell = \lceil n/16 \rceil$. This means the last block is incomplete. This last block is handled as stated in [5] and the computation follows the said methods.

It can be said that 4-decimation Horner [7] is followed up to the length where number of 16-byte blocks is multiple of 4. Then one of the above discussed methods is followed.

Example:

- If after applying 4-decimation Horner on input message still 63 more bytes are left, these 63 bytes are broken into 3 parts: 32 bytes, 16 bytes, 15 bytes. Thus, fresh alignments of the remaining data are done 3 times.
- If after applying 4-decimation Horner on input message still 40 more bytes are left, these 40 bytes are broken into 2 parts: 32 bytes, 8 bytes. Thus, fresh alignments of the remaining data are done 2 times.

3.4 Description of the Goll-Gueron 4x130 multiplication algorithm

`Input`: Padded and converted 64 bytes of the input message in 3 256-bit registers, R^4 and $5 \cdot R^4$ aligned in 2 256-bit registers. All the alignments are done as discussed earlier.
`Output`: The output is stored in 5 256-bit registers.

t_{30}	t_{20}	t_{10}	t_{00}

t_{31}	t_{21}	t_{11}	t_{01}

t_{32}	t_{22}	t_{12}	t_{02}

t_{33}	t_{23}	t_{13}	t_{03}

t_{34}	t_{24}	t_{14}	t_{04}

Note 1: Thus, all the registers used for this purpose are 256-bit long. Operands are divided into 8 32-byte parts and the products are divided into 4 64-byte parts.
Note 2: A brief summary of the intrinsics used in this algorithm:
Step 1:
To obtain partial results of t_{00}, t_{10}, t_{20} and t_{30}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	t_{32}	t_{30}	t_{22}	t_{20}	t_{12}	t_{10}	t_{02}	t_{00}
operand 2:	g_1	g_0	g_1	g_0	g_1	g_0	g_1	g_0

Product:	$t_{30} \cdot g_0$	$t_{20} \cdot g_0$	$t_{10} \cdot g_0$	$t_{00} \cdot g_0$

Step 2:
To obtain partial results of t_{01}, t_{11}, t_{21} and t_{31}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	t_{33}	t_{31}	t_{23}	t_{21}	t_{13}	t_{11}	t_{03}	t_{01}
operand 2:	g_1	g_0	g_1	g_0	g_1	g_0	g_1	g_0

Product:	$t_{31} \cdot g_0$	$t_{21} \cdot g_0$	$t_{11} \cdot g_0$	$t_{01} \cdot g_0$

Step 3:
To obtain partial results of t_{02}, t_{12}, t_{22} and t_{32}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	x	t_{34}	x	t_{24}	x	t_{14}	x	t_{04}
operand 2:	g_1	g_0	g_1	g_0	g_1	g_0	g_1	g_0

Product:	$t_{34} \cdot g_0$	$t_{24} \cdot g_0$	$t_{14} \cdot g_0$	$t_{04} \cdot g_0$

Step 4:
To obtain partial results of t_{04}, t_{14}, t_{24} and t_{34}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	t_{32}	t_{30}	t_{22}	t_{20}	t_{12}	t_{10}	t_{02}	t_{00}
operand 2:	g_3	g_2	g_3	g_2	g_3	g_2	g_3	g_2

product:	$t_{30} \cdot g_2$	$t_{20} \cdot g_2$	$t_{10} \cdot g_2$	$t_{00} \cdot g_2$

Step 5:
To obtain partial results of t_{03}, t_{13}, t_{23} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:

t_{33}	t_{31}	t_{23}	t_{21}	t_{13}	t_{11}	t_{03}	t_{01}

operand 2:

g_3	g_2	g_3	g_2	g_3	g_2	g_3	g_2

Product:

$t_{31} \cdot g_2$	$t_2 \cdot g_2$	$t_1 \cdot g_2$	$t_{01} \cdot g_2$

Step 6:
To obtain partial results of t_{01}, t_{11}, t_{21} and t_{31}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:

t_{32}	t_{30}	t_{22}	t_{20}	t_{12}	t_{10}	t_{02}	t_{00}

operand 2:

g_0	g_1	g_0	g_1	g_0	g_1	g_0	g_1

product:

$t_{30} \cdot g_1$	$t_{20} \cdot g_1$	$t_{10} \cdot g_1$	$t_{00} \cdot g_1$

Add this product with result obtained from Step 2 to obtain:

$t_{31} \cdot g_0 +$ $t_{30} \cdot g_1$	$t_{20} \cdot g_0 +$ $t_{21} \cdot g_1$	$t_{11} \cdot g_0 +$ $t_{10} \cdot g_1$	$t_{01} \cdot g_0 +$ $t_{00} \cdot g_1$

Step 7:
To obtain partial results of t_{02}, t_{12}, t_{22} and t_{32}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:

t_{33}	t_{31}	t_{23}	t_{21}	t_{13}	t_{11}	t_{03}	t_{01}

operand 2:

g_0	g_1	g_0	g_1	g_0	g_1	g_0	g_1

product:

$t_{31} \cdot g_1$	$t_{21} \cdot g_1$	$t_{11} \cdot g_1$	$t_{01} \cdot g_1$

Add this product with the results obtained in Step 4 to obtain:

$t_{30} \cdot g_2 +$ $t_{31} \cdot g_1$	$t_{20} \cdot g_2 +$ $t_{21} \cdot g_1$	$t_{10} \cdot g_2 +$ $t_{11} \cdot g_1$	$t_{00} \cdot g_2 +$ $t_{01} \cdot g_1$

Step 8:
To obtain partial results of t_{03}, t_{13}, t_{23} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:

t_{32}	t_{30}	t_{22}	t_{20}	t_{12}	t_{10}	t_{02}	t_{00}

operand 2:

g_2	g_3	g_2	g_3	g_2	g_3	g_2	g_3

product:

$t_{30} \cdot g_3$	$t_{20} \cdot g_3$	$t_{10} \cdot g_3$	$t_{00} \cdot g_3$

Add this product with result obtained from Step 5 to get:

$t_{31} \cdot g_2 +$ $t_{30} \cdot g_3$	$t_{21} \cdot g_2 +$ $t_{20} \cdot g_3$	$t_{11} \cdot g_2 +$ $t_{10} \cdot g_3$	$t_{01} \cdot g_2 +$ $t_{00} \cdot g_3$

Step 9:
To obtain partial results of t_{04}, t_{14}, t_{24} and t_{34}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:

t_{33}	t_{31}	t_{23}	t_{21}	t_{13}	t_{11}	t_{03}	t_{01}

operand 2:

g_2	g_3	g_2	g_3	g_2	g_3	g_2	g_3

product:

$t_{31} \cdot g_3$	$t_{21} \cdot g_3$	$t_{11} \cdot g_3$	$t_{01} \cdot g_3$

Add this product with result obtained in Step 3 to get:

$t_{34} \cdot g_0 +$ $t_{31} \cdot g_3$	$t_{24} \cdot g_0 +$ $t_{21} \cdot g_3$	$t_{14} \cdot g_0 +$ $t_{11} \cdot g_3$	$t_{04} \cdot g_0 +$ $t_{01} \cdot g_3$

Step 10:
To obtain partial results of t_{00}, t_{10}, t_{20} and t_{30}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:

t_{32}	t_{30}	t_{22}	t_{20}	t_{12}	t_{10}	t_{02}	t_{00}

operand 2:

$5 \cdot g_1$	g_4	$5 \cdot g_1$	g_1	$5 \cdot g_1$	g_4	$5 \cdot g_1$	g_4

product:

$t_{30} \cdot g_4$	$t_{20} \cdot g_4$	$t_{10} \cdot g_4$	$t_{00} \cdot g_4$

Add this product with result obtained in Step 9:

$t_{34} \cdot g_0 +$ $t_{31} \cdot g_3 +$ $t_{30} \cdot g_4$	$t_{24} \cdot g_0 +$ $t_{21} \cdot g_3 +$ $t_{20} \cdot g_4$	$t_{14} \cdot g_0 +$ $t_{11} \cdot g_3 +$ $t_{10} \cdot g_4$	$t_{04} \cdot g_0 +$ $t_{01} \cdot g_3 +$ $t_{00} \cdot g_4$

Till now only the lower order 32 bits of each packed 64 bits of the 256-bit long registers were multiplied with the corresponding lower order 32 bits of each packed 64 bits of the 256-bit long registers containing R^4. Now, the multiplications on the higher order 32 bits of each packed 64 bits of the 256-bit long registers with the corresponding lower order 32 bits of each packed 64 bits of the 256-bit long registers containing R^4 need to be done. So the lower and upper order 32 bits of each packed 64 bits of those registers are swapped.

Step 11:
To obtain partial results of t_{00}, t_{10}, t_{20} and t_{30}.
Operation involved: *_mm256_mul_epu32* and *_mm256_add_epi64*

operand 1:	t_{31}	t_{33}	t_{21}	t_{23}	t_{11}	t_{13}	t_{01}	t_{03}
operand 2:	g_4	$5 \cdot g_1$	g_4	$5 \cdot g_1$	g_4	$5 \cdot g_1$	g_4	$5 \cdot g_1$

product:	$5 \cdot g_1 \cdot t_{33}$	$5 \cdot g_1 \cdot t_{23}$	$5 \cdot g_1 \cdot t_{13}$	$5 \cdot g_1 \cdot t_{03}$

Add this product with the result obtained from Step 1

$t_{30} \cdot g_0 + 5 \cdot g_1 \cdot t_{33}$	$5 \cdot g_1 \cdot t_{23} + t_{20} \cdot g_0$	$t_{10} \cdot g_0 + 5 \cdot g_1 \cdot t_{13}$	$t_{00} \cdot g_0 + 5 \cdot g_1 \cdot t_{03}$

Step 12:
To obtain partial results of t_{04}, t_{14}, t_{24} and t_{34}.
Operation involved: *_mm256_mul_epu32* and *_mm256_add_epi64*

operand 1:	x	t_{34}	x	t_{24}	x	t_{14}	x	t_{04}
operand 2:	g_4	$5 \cdot g_1$	g_4	$5 \cdot g_1$	g_4	$5 \cdot g_1$	g_4	$5 \cdot g_1$

product:	$5 \cdot g_1 \cdot t_{34}$	$5 \cdot g_1 \cdot t_{24}$	$5 \cdot g_1 \cdot t_{14}$	$5 \cdot g_1 \cdot t_{04}$

Add this product to result obtained from Step 6.

$t_{31} \cdot g_0 + t_{30} \cdot g_1 + 5 \cdot g_1 \cdot t_{34}$	$t_{21} \cdot g_0 + t_{20} \cdot g_1 + 5 \cdot g_1 \cdot t_{24}$	$t_{11} \cdot g_0 + t_{10} \cdot g_1 + 5 \cdot g_1 \cdot t_{14}$	$t_{01} \cdot g_0 + t_{00} \cdot g_1 + 5 \cdot g_1 \cdot t_{04}$

Step 13:
To obtain partial results of t_{32}, t_{22}, t_{12} and t_{02}.
Operation involved: *_mm256_mul_epu32* and *_mm256_add_epi64*

operand 1:	t_{30}	t_{32}	t_{20}	t_{22}	t_{10}	t_{12}	t_{00}	t_{02}
operand 2:	g_0	g_1	g_0	g_1	g_0	g_1	g_0	g_1

product:	$t_{32} \cdot g_1$	$t_{22} \cdot g_1$	$t_{12} \cdot g_1$	$t_{02} \cdot g_1$

Add this product with the result obtained from Step 8:

$t_{31} \cdot g_2 + t_{30} \cdot g_3 + t_{32} \cdot g_1$	$t_{21} \cdot g_2 + t_{20} \cdot g_3 + t_{22} \cdot g_1$	$t_{11} \cdot g_2 + t_{10} \cdot g_3 + t_{12} \cdot g_1$	$t_{01} \cdot g_2 + t_{00} \cdot g_3 + t_{02} \cdot g_1$

Step 14:
To obtain partial results of t_{33}, t_{23}, t_{13} and t_{03}.
Operation involved: _mm256_mul_epu32 and _mm256_add_epi64

operand 1:	t_{31}	t_{33}	t_{21}	t_{23}	t_{11}	t_{13}	t_{01}	t_{03}
operand 2:	g_0	g_1	g_0	g_1	g_0	g_1	g_0	g_1

product:	$t_{33} \cdot g_1$	$t_{23} \cdot g_1$	$t_{13} \cdot g_1$	$t_{03} \cdot g_1$

Add this product with the result from Step 10

$t_{34} \cdot g_0 + t_{31} \cdot g_3 + t_{30} \cdot g_4 + t_{33} \cdot g_1$	$t_{24} \cdot g_0 + t_{21} \cdot g_3 + t_{20} \cdot g_4 + t_{23} \cdot g_1$	$t_{14} \cdot g_0 + t_{11} \cdot g_3 + t_{10} \cdot g_4 + t_{13} \cdot g_1$	$t_{04} \cdot g_0 + t_{01} \cdot g_3 + t_{00} \cdot g_4 + t_{03} \cdot g_1$

Step 15:
To obtain partial results of t_{32}, t_{22}, t_{12} and t_{02}.
Operation involved: _mm256_mul_epu32 and _mm256_add_epi64

operand 1:	t_{30}	t_{32}	t_{20}	t_{22}	t_{10}	t_{12}	t_{00}	t_{02}
operand 2:	g_1	g_0	g_1	g_0	g_1	g_0	g_1	g_0

product:	$t_{32} \cdot g_0$	$t_{22} \cdot g_0$	$t_{12} \cdot g_0$	$t_{02} \cdot g_0$

Add the result obtained from Step 7

$t_{30} \cdot g_2 + t_{31} \cdot g_1 + t_{32} \cdot g_0$	$t_{20} \cdot g_2 + t_{21} \cdot g_1 + t_{22} \cdot g_0$	$t_{10} \cdot g_2 + t_{11} \cdot g_1 + t_{12} \cdot g_0$	$t_{00} \cdot g_2 + t_{01} \cdot g_1 + t_{02} \cdot g_0$

Step 16:
To obtain partial results of t_{32}, t_{22}, t_{12} and t_{02}.
Operation involved: _mm256_mul_epu32 and _mm256_add_epi64

operand 1:	t_{31}	t_{33}	t_{21}	t_{23}	t_{11}	t_{13}	t_{01}	t_{03}
operand 2:	g_1	g_0	g_1	g_0	g_1	g_0	g_1	g_0

product:	$t_{33} \cdot g_0$	$t_{23} \cdot g_0$	$t_{13} \cdot g_0$	$t_{03} \cdot g_0$

Add this product with the result obtained from Step 13:

$t_{31} \cdot g_2 + t_{30} \cdot g_3 + t_{32} \cdot g_1 + t_{33} \cdot g_0$	$t_{21} \cdot g_2 + t_{20} \cdot g_3 + t_{22} \cdot g_1 + t_{23} \cdot g_0$	$t_{11} \cdot g_2 + t_{10} \cdot g_3 + t_{12} \cdot g_1 + t_{13} \cdot g_0$	$t_{01} \cdot g_2 + t_{00} \cdot g_3 + t_{02} \cdot g_1 + t_{03} \cdot g_0$

Step 17:
To obtain partial results of t_{32}, t_{22}, t_{12} and t_{02}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:	t_{30}	t_{32}	t_{20}	t_{22}	t_{10}	t_{12}	t_{00}	t_{02}
operand 2:	g_3	g_2	g_3	g_2	g_3	g_2	g_3	g_2

product:	$t_{32} \cdot g_2$	$t_{22} \cdot g_2$	$t_{12} \cdot g_2$	$t_{02} \cdot g_2$

Add result obtained from Step 14:

$t_{34} \cdot g_0 + t_{31} \cdot g_3 + t_{30} \cdot g_4 + t_{33} \cdot g_1 + t_{32} \cdot g_2$	$t_{24} \cdot g_0 + t_{21} \cdot g_3 + t_{20} \cdot g_4 + t_{23} \cdot g_1 + t_{22} \cdot g_2$	$t_{14} \cdot g_0 + t_{11} \cdot g_3 + t_{10} \cdot g_4 + t_{13} \cdot g_1 + t_{12} \cdot g_2$	$t_{04} \cdot g_0 + t_{01} \cdot g_3 + t_{00} \cdot g_4 + t_{03} \cdot g_1 + t_{02} \cdot g_2$

Step 18:
To obtain partial results of t_{32}, t_{22}, t_{12} and t_{02}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:	t_{30}	t_{32}	t_{20}	t_{22}	t_{10}	t_{12}	t_{00}	t_{02}
operand 2:	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$

product:	$5 \cdot g_2 \cdot t_{32}$	$5 \cdot g_2 \cdot t_{22}$	$5 \cdot g_2 \cdot t_{12}$	$5 \cdot g_2 \cdot t_{02}$

Add this product with the result obtained from Step 11:

$t_{30} \cdot g_0 + 5 \cdot g_1 \cdot t_{33} + t_{32} \cdot g_3$	$5 \cdot g_1 \cdot t_{23} + t_{20} \cdot g_0 + t_{22} \cdot g_3$	$t_{10} \cdot g_0 + 5 \cdot g_1 \cdot t_{13} + t_{12} \cdot g_3$	$t_{00} \cdot g_0 + 5 \cdot g_1 \cdot t_{03} + t_{02} \cdot g_3$

Step 19:
To obtain partial results of t_{33}, t_{23}, t_{13} and t_{03}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:	t_{31}	t_{33}	t_{21}	t_{23}	t_{11}	t_{13}	t_{01}	t_{03}
operand 2:	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$

product:	$5 \cdot g_2 \cdot t_{33}$	$5 \cdot g_2 \cdot t_{23}$	$5 \cdot g_2 \cdot t_{13}$	$5 \cdot g_2 \cdot t_{03}$

Add this product with the result obtained from Step 12:

$t_{31} \cdot g_0 + t_{30} \cdot g_1 + 5 \cdot g_1 \cdot t_{34} + 5 \cdot g_2 \cdot t_{33}$	$t_{21} \cdot g_0 + t_{20} \cdot g_1 + 5 \cdot g_1 \cdot t_{24} + 5 \cdot g_2 \cdot t_{23}$	$t_{11} \cdot g_0 + t_{10} \cdot g_1 + 5 \cdot g_1 \cdot t_{14} + 5 \cdot g_2 \cdot t_{13}$	$t_{01} \cdot g_0 + t_{00} \cdot g_1 + 5 \cdot g_1 \cdot t_{04} + 5 \cdot g_2 \cdot t_{03}$

Step 20:
To obtain partial results of t_{34}, t_{24}, t_{14} and t_{04}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:	x	t_{34}	x	t_{24}	x	t_{14}	x	t_{04}
operand 2:	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$

product:	$5 \cdot g_2 \cdot t_{34}$	$5 \cdot g_2 \cdot t_{24}$	$5 \cdot g_2 \cdot t_{14}$	$5 \cdot g_2 \cdot t_{04}$

Add this product with the result obtained from Step 15:

$t_{30} \cdot g_2 + t_{31} \cdot g_1 + t_{32} \cdot g_0 + 5 \cdot g_2 \cdot t_{34}$	$t_{20} \cdot g_2 + t_{21} \cdot g_1 + t_{22} \cdot g_0 + 5 \cdot g_2 \cdot t_{24}$	$t_{10} \cdot g_2 + t_{11} \cdot g_1 + t_{12} \cdot g_0 + 5 \cdot g_2 \cdot t_{14}$	$t_{00} \cdot g_2 + t_{01} \cdot g_1 + t_{02} \cdot g_0 + 5 \cdot g_2 \cdot t_{04}$

Step 21:
To obtain partial results of t_{34}, t_{24}, t_{14} and t_{04}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:	x	t_{34}	x	t_{24}	x	t_{14}	x	t_{04}
operand 2:	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$

product:	$5 \cdot g_2 \cdot t_{34}$	$5 \cdot g_2 \cdot t_{24}$	$5 \cdot g_2 \cdot t_{14}$	$5 \cdot g_2 \cdot t_{04}$

Add this product with result obtained from Step 16:

$t_{31} \cdot g_2 + t_{30} \cdot g_3 + t_{32} \cdot g_1 + t_{33} \cdot g_0 + 5 \cdot g_2 \cdot t_{34}$	$t_{21} \cdot g_2 + t_{20} \cdot g_3 + t_{22} \cdot g_1 + t_{23} \cdot g_0 + 5 \cdot g_2 \cdot t_{24}$	$t_{11} \cdot g_2 + t_{10} \cdot g_3 + t_{12} \cdot g_1 + t_{13} \cdot g_0 + 5 \cdot g_2 \cdot t_{14}$	$t_{01} \cdot g_2 + t_{00} \cdot g_3 + t_{02} \cdot g_1 + t_{03} \cdot g_0 + 5 \cdot g_2 \cdot t_{04}$

Step 22:
To obtain partial results of t_{32}, t_{22}, t_{12} and t_{02}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:	t_{30}	t_{32}	t_{20}	t_{22}	t_{10}	t_{12}	t_{00}	t_{02}
operand 2:	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$

product:	$5 \cdot g_2 \cdot t_{32}$	$5 \cdot g_2 \cdot t_{22}$	$5 \cdot g_2 \cdot t12$	$5 \cdot g_2 \cdot t_{02}$

Add this product with the result obtained from Step 19:

$t_{31} \cdot g_0 + t_{30} \cdot g_1 + 5 \cdot g_1 \cdot t_{34} + 5 \cdot g_2 \cdot t_{33} + 5 \cdot g_2 \cdot t_{32}$	$t_{21} \cdot g_0 + t_{20} \cdot g_1 + 5 \cdot g_1 \cdot t_{24} + 5 \cdot g_2 \cdot t_{23} + 5 \cdot g_2 \cdot t_{22}$	$t_{11} \cdot g_0 + t_{10} \cdot g_1 + 5 \cdot g_1 \cdot t_{14} + 5 \cdot g_2 \cdot t_{13} + 5 \cdot g_2 \cdot t12$	$t_{01} \cdot g_0 + t_{00} \cdot g_1 + 5 \cdot g_1 \cdot t_{04} + 5 \cdot g_2 \cdot t_{03} + 5 \cdot g_2 \cdot t_{02}$

Step 23:
To obtain partial results of $t_{33},\ t_{23},\ t_{13}$ and t_{03}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:	t_{31}	t_{33}	t_{21}	t_{23}	t_{11}	t_{13}	t_{01}	t_{03}
operand 2:	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$

product:	$5 \cdot g_2 \cdot t_{33}$	$5 \cdot g_2 \cdot t_{23}$	$5 \cdot g_2 \cdot t_{13}$	$5 \cdot g_2 \cdot t_{03}$

Add this product to the result obtained from Step 21:

$t_{31} \cdot g_2 + t_{30} \cdot g_3 + t_{32} \cdot g_1 + t_{33} \cdot g_0 + 5 \cdot g_2 \cdot t_{34} + 5 \cdot g_2 \cdot t_{33}$	$t_{21} \cdot g_2 + t_{20} \cdot g_3 + t_{22} \cdot g_1 + t_{23} \cdot g_0 + 5 \cdot g_2 \cdot t_{24} + 5 \cdot g_2 \cdot t_{23}$	$t_{11} \cdot g_2 + t_{10} \cdot g_3 + t_{12} \cdot g_1 + t_{13} \cdot g_0 + 5 \cdot g_2 \cdot t_{14} + 5 \cdot g_2 \cdot t_{13}$	$t_{01} \cdot g_2 + t_{00} \cdot g_3 + t_{02} \cdot g_1 + t_{03} \cdot g_0 + 5 \cdot g_2 \cdot t_{04} + 5 \cdot g_2 \cdot t_{03}$

Step 24:
To obtain partial results of $t_{31},\ t_{21},\ t_{11}$ and t_{01}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:	t_{33}	t_{31}	t_{23}	t_{21}	t_{13}	t_{11}	t_{03}	t_{01}
operand 2:	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_3$	$5 \cdot g_2$

product:	$5 \cdot g_3 \cdot t_{31}$	$5 \cdot g_3 \cdot t_{21}$	$5 \cdot g_3 \cdot t_{11}$	$5 \cdot g_3 \cdot t_{01}$

Add this product to the result obtained from Step 18:

$t_{30} \cdot g_0 + 5 \cdot g_1 \cdot t_{33} + t_{32} \cdot g_3 + 5 \cdot g_3 \cdot t_{31}$	$5 \cdot g_1 \cdot t_{23} + t_{20} \cdot g_0 + t_{22} \cdot g_3 + 5 \cdot g_3 \cdot t_{21}$	$t_{10} \cdot g_0 + 5 \cdot g_1 \cdot t_{13} + t_{12} \cdot g_3 + 5 \cdot g_3 \cdot t_{11}$	$t_{00} \cdot g_0 + 5 \cdot g_1 \cdot t_{03} + t_{02} \cdot g_3 + 5 \cdot g_3 \cdot t_{01}$

Step 25:
To obtain partial results of $t_{34},\ t_{24},\ t_{14}$ and t_{04}.
Operation involved: $\_mm256\_mul\_epu32$ and $\_mm256\_add\_epi64$

operand 1:

x	t_{34}	x	t_{24}	x	t_{14}	x	t_{04}

operand 2:

$5 \cdot g_0$	$5 \cdot g_0$	$5 \cdot g_0$	$5 \cdot g_0$	$5 \cdot g_0$	$5 \cdot g_0$	$5 \cdot g_0$	$5 \cdot g_0$

product:

$5 \cdot g_0 \cdot t_{34}$	$5 \cdot g_0 \cdot t_{24}$	$5 \cdot g_0 \cdot t_{14}$	$5 \cdot g_0 \cdot t_{04}$

Add this product to the result obtained from Step 24

$t_{30} \cdot g_0 + 5 \cdot g_1 \cdot t_{33} + t_{32} \cdot g_3 + 5 \cdot g_3 \cdot t_{31} + 5 \cdot g_0 \cdot t_{34}$	$5 \cdot g_1 \cdot t_{23} + t_{20} \cdot g_0 + t_{22} \cdot g_3 + 5 \cdot g_3 \cdot t_{21} + 5 \cdot g_0 \cdot t_{24}$	$t_{10} \cdot g_0 + 5 \cdot g_1 \cdot t_{13} + t_{12} \cdot g_3 + 5 \cdot g_3 \cdot t_{11} + 5 \cdot g_0 \cdot t_{14}$	$t_{00} \cdot g_0 + 5 \cdot g_1 \cdot t_{03} + t_{02} \cdot g_3 + 5 \cdot g_3 \cdot t_{01} + 5 \cdot g_0 \cdot t_{04}$

Step 26:

End

After this multiplication operation, we need a reduction operation so that the result can be stored in 3 256-bit registers i.e, each of the 64-bit products are reduced to fit in 32-bit registers.

Operation Count for the 4×130 multiplication algorithm

Name of Intrinsic	Count
_mm256_mul_epu32	25
_mm256_set_epi32	1
_mm256_add_epi64	20
_mm256_permutevar8x32_epi32	9
_mm256_permute4x64_epi64	4

3.5 Description of Goll-Gueron SIMD Reduction From 64 bits to 32 bits

Let us consider any $i \in \{0, 1, 2, 3\}$. The corresponding $(t_{i0}, t_{i1}, t_{i2}, t_{i3}, t_{i4})$ is reduced using and interleaving the following 2 independent chains: $t_{i0} \rightarrow t_{i1} \rightarrow t_{i2} \rightarrow t_{i3} \rightarrow t_{i4}$ and $t_{i3} \rightarrow t_{i4} \rightarrow t_{i0} \rightarrow t_{i1}$. The basic concept is explained by Bernstein in [6]. The computation is as follows:

$$\left.\begin{array}{l}
1.\ t_{i1} = t_{i1} + \lfloor t_{i0}/2^{26} \rfloor,\ t_{i0} = t_{i0} \mod 2^{26} \\[1em]
2.\ t_{i4} = t_{i4} + \lfloor t_{i3}/2^{26} \rfloor,\ t_{i3} = t_{i3} \mod 2^{26} \\[1em]
3.\ t_{i2} = t_{i2} + \lfloor t_{i1}/2^{26} \rfloor,\ t_{i1} = t_{i1} \mod 2^{26} \\[1em]
4.\ t_{i0} = t_{i0} + 5 \cdot \lfloor t_{i4}/2^{26} \rfloor,\ t_{i4} = t_{i4} \mod 2^{26} \\[1em]
5.\ t_{i3} = t_{i3} + \lfloor t_{i2}/2^{26} \rfloor,\ t_{i2} = t_{i2} \mod 2^{26} \\[1em]
6.\ t_{i1} = t_{i1} + \lfloor t_{i0}/2^{26} \rfloor,\ t_{i0} = t_{i0} \mod 2^{26} \\[1em]
7.\ t_{i4} = t_{i4} + \lfloor t_{i3}/2^{26} \rfloor,\ t_{i3} = t_{i3} \mod 2^{26}
\end{array}\right\} \text{..... B}$$

This reduction alternates between the two chains. Using the above concept Goll and Gueron have proposed the following algorithm.

The 4×130 **reduction algorithm by Goll-Gueron**

`Input:` The product obtained from the 4×130 multiplication algorithm contained in 5 256-bit registers.

t_{30}	t_{20}	t_{10}	t_{00}

t_{31}	t_{21}	t_{11}	t_{01}

t_{32}	t_{22}	t_{12}	t_{02}

t_{33}	t_{23}	t_{13}	t_{03}

`Output:` Reduced form the input product stored in 3 256-bit registers.
Note: A brief summary of the intrinsics [2] used in this algorithm:

- *_mm256_add_epi64* (*__m256i a*, *__m256i b*): Add packed 64-bit integers in a and b, and store the results in dst.
- *_mm256_srli_epi64* (*_m256i a*, *int imm8*): Shift packed 64-bit integers in a right by imm8 while shifting in zeros, and store the results in dst.
- *_mm256_and_si256* (*__m256i a*, *__m256i b*): Compute the bitwise AND of 256 bits (representing integer data) in a and b, and store the result in dst.
- *_mm256_blend_epi32* (*__m256i a*, *__m256i b*, *const int imm8*): Blend packed 32-bit integers from a and b using control mask imm8, and store the results in dst.
- *_mm256_slli_epi64* (*__m256i a*, *int imm8*): Shift packed 64-bit integers in a left by imm8 while shifting in zeros, and store the results in dst.

1. Generate mask such that only lower order 26 bits of each packed 64-bit of a 256-bit register are set.Thus, mask is stored in a 256-bit register.
   mask $\leftarrow$ $(vec256)\,\{$`0x3ffffff,0,0x3ffffff,0,0x3ffffff,0,0x3ffffff,0`$\}$

2. */*Computing the first equation of the system B*/*
   x.v1=$\_mm256\_add\_epi64\,(x.v1,\ \_mm256\_srli\_epi64\,(x.v_0, 26)))$
   x.v0=$\_mm256\_and\_si256\,(x.v0, mask)$

3. */*Computing the second equation of the system B*/*
   x.v4=$\_mm256\_add\_epi64\,(x.v4,\ \_mm256\_srli\_epi64\,(x.v3, 26)))$
   x.v3=$\_mm256\_and\_si256\,(x.v3, mask)$

4. */*Computing the third equation of the system B*/*
   x.v2=$\_mm256\_add\_epi64\,(x.v2,\ \_mm256\_srli\_epi64\,(x.v1, 26)))$
   x.v1=$\_mm256\_and\_si256\,(x.v1, mask)$

5. */*Computing the fourth equation of the system B*/*
   x.v0=$\_mm256\_add\_epi64\,(x.v0, \_mm256\_mul\_epu32\,(\_mm256\_srli\_epi64\,(x.v4, 26)))\,, 5)$
   x.v4=$\_mm256\_and\_si256\,(x.v4, mask)$

6. */*Computing the fifth equation of the system B*/*
   x.v3=$\_mm256\_add\_epi64\,(x.v3,\ \_mm256\_srli\_epi64\,(x.v2, 26)))$
   x.v2=$\_mm256\_and\_si256\,(x.v2, mask)$

7. */*Computing the sixth equation of the system B*/*
   x.v1=$\_mm256\_add\_epi64\,(x.v1,\ \_mm256\_srli\_epi64\,(x.v0, 26)))$
   x.v0=$\_mm256\_and\_si256\,(x.v0, mask)$

8. */*Computing the seventh equation of the system B*/*
   x.v4=$\_mm256\_add\_epi64\,(x.v4,\ \_mm256\_srli\_epi64\,(x.v3, 26)))$
   /*This x.v4 is the 3rd register of the required alignment*/
   x.v3=$\_mm256\_and\_si256\,(x.v3, mask)$
   /*At this point all the reductions are complete.Now it is required to achieve the required alignment. Since we have obtained the third register, we need to find the other two registers*/

9. x.v0 = $\_mm256\_blend\_epi32\,(x.v0,\ \_mm256\_slli\_epi64\,(x.v2,\ 32)\,,$ `0xAA`$)$/*content of the first 256-bit register*/

10. x.v1 = $\_mm256\_blend\_epi32\,(x.v1,\ \_mm256\_slli\_epi64\,(x.v3,\ 32)\,,$ `0xAA`$)$/*content of the second 256-bit register*/

4 Our Contribution: Improved 256-bit Vectorization of Poly1305

4.1 Revisiting 4-decimation Horner

Given a sequence of ℓ 130-bit blocks $M_0, ..., M_{\ell-1}$, $\mathbf{Horner}_R\,(M_0, ..., M_{\ell-1})$ can be computed in the following way which is termed as decimated Horner [7].

$$\begin{aligned}\mathbf{Horner}_R\left(M_0,...,M_{\ell-1}\right) &= R^4\mathbf{Horner}_{R^4}\left(M_0,M_4,...,M_{4i},...\right)\\ &\quad +R^3\mathbf{Horner}_{R^4}\left(M_1,M_5,...,M_{4i+1},...\right)\\ &\quad +R^2\mathbf{Horner}_{R^4}\left(M_2,M_6,...,M_{4i+2},...\right)\\ &\quad +R\mathbf{Horner}_{R^4}\left(M_3,M_7,...,M_{4i+3},...\right)\end{aligned}$$

4.2 Partitioning the message space

In [5] it is clearly stated that if a message has n bytes, then it has $\lceil n/16 \rceil$ 16-byte blocks. But the implementation [9] considers $n/16$ 16-byte blocks. If $16 \not| n$, due to the presence of an incomplete last block, fresh packings are required. This may require multiple fresh packings depending on the length of the message. In the improved version of 256-bit vectorization of Poly1305, we follow the a variant of 4-decimation Horner for computation irrespective of the message length and thus the message space can be partitioned into the following four disjoint cases.

- case 0: messages where number of 16-byte blocks $\equiv 0 \bmod 4$
- case 1: messages where number of 16-byte blocks $\equiv 1 \bmod 4$
- case 2: messages where number of 16-byte blocks $\equiv 2 \bmod 4$
- case 3: messages where number of 16-byte blocks $\equiv 3 \bmod 4$

4.3 Modified 4-decimation Horner for Optimized Evaluation

We get scope for modifying the 4-decimation Horner due to the 8 cases mentioned in the previous section. The repeated fresh packings that are required by [8] waste some machine cycles. So we use a variant of 4-decimation Horner.

Various forms of Modified 4-decimation Horner's Rule for optimized evaluation

$1 \leq$ **message text length in bytes left after 4-decimation** $\leq$ **15**. In this case the incomplete 16-byte block is appended with 1-bit and required number of 0 bits as stated in [5] and 48 zeros are prepended logically. Then the modified 4-decimation Horner is followed as in the next case.

message text length in bytes left after 4-decimation = 16. The modified input format for the optimized version is: $0 \cdot R^{\ell+3} + 0 \cdot R^{\ell+2} + 0 \cdot R^{\ell+1} + c_0 \cdot R^{\ell} + \ldots + c_{\ell-1} \cdot R$. Here 48 zeros are prepended logically. So the 4-decimation Horner for this modified form of input looks like:

$$\begin{aligned}\mathbf{Horner}_R\left(c_0,...,c_{\ell-1}\right) &= R^4\ \mathbf{Horner}_{R^4}\left(0,c_1,...,c_{4i-3},...\right)\\ &\quad +R^3\ \mathbf{Horner}_{R^4}\left(0,c_2,...,c_{4i-2},...\right)\\ &\quad +R^2\ \mathbf{Horner}_{R^4}\left(0,c_3,...,c_{4i-1},...\right)\\ &\quad +R\ \mathbf{Horner}_{R^4}\left(c_0,c_4,...,c_{4i},...\right)\end{aligned}$$

17 $\leq$ message text length in bytes left after 4-decimation $\leq$ 31. In this case the incomplete 16-byte block is appended with 1-bit and required number of 0 bits as stated in [5] and 32 zeros are prepended logically. Then the modified 4-decimation Horner is followed as in the next case.

message text length in bytes left after 4-decimation = 32. The modified input format for the optimized version is: $0 \cdot R^{\ell+2} + 0 \cdot R^{\ell+1} + c_0 \cdot R^{\ell} + \ldots + c_{\ell-1} \cdot R$. Thus 32 zeros are prepended logically. So the 4-decimation Horner for this modified form of input looks like:

$$\begin{aligned} \textbf{Horner}_R\left(c_0, ..., c_{\ell-1}\right) &= R^4 \ \textbf{Horner}_{R^4}\left(0, c_2, ..., c_{4i-2}, ...\right) \\ &\quad + R^3 \ \textbf{Horner}_{R^4}\left(0, c_3, ..., c_{4i-1}, ...\right) \\ &\quad + R^2 \ \textbf{Horner}_{R^4}\left(c_0, c_4, ..., c_{4i}, ...\right) \\ &\quad + R \ \textbf{Horner}_{R^4}\left(c_1, c_5, ..., c_{4i+1}, ...\right) \end{aligned}$$

33 $\leq$ message text length in bytes left after 4-decimation $\leq$ 47. In this case the incomplete 16-byte block is appended with 1-bit and required number of 0 bits as stated in [5] and 16 zeros are prepended logically. Then the modified 4-decimation Horner is followed as in the next case.

message text length in bytes left after 4-decimation = 48. The modified input format for the optimized version is: $0 \cdot R^{\ell+1} + c_0 \cdot R^{\ell} + \ldots + c_{\ell-1} \cdot R$. So the 4-decimation Horner for this modified form of input looks like:

$$\begin{aligned} \textbf{Horner}_R\left(c_0, ..., c_{\ell-1}\right) &= R^4 \ \textbf{Horner}_{R^4}\left(0, c_3, ..., c_{4i-1}, ...\right) \\ &\quad + R^3 \ \textbf{Horner}_{R^4}\left(c_0, c_4, ..., c_{4i}, ...\right) \\ &\quad + R^2 \ \textbf{Horner}_{R^4}\left(c_1, c_5, ..., c_{4i+1}, ...\right) \\ &\quad + R \ \textbf{Horner}_{R^4}\left(c_2, c_6, ..., c_{4i+2}, ...\right) \end{aligned}$$

Logically 16 zeros are prepended. Then the algorithm for case 0 as stated in this section 4 is followed for evaluation. As stated in [5] the each 16-byte block of the input message should be concatenated with 1-bit. But in this case the zeros prepended are not part of the real message, so no 1-bit should be concatenated.

49 $\leq$ message text length in bytes left after 4-decimation $\leq$ 63. In this case the incomplete 16-byte block is appended with 1-bit and required number of 0 bits as stated in [5] and then 4-decimation Horner is followed as given in Section 4.1.

4.4 Modified 4x130 multiplication algorithm for messages belonging to case 1

For case 1 in Section 3.3, i.e., $n' = 16$, the alignment of the message is:

t_{32}	t_{30}	0	0	0	0	0	0
t_{33}	t_{31}	0	0	0	0	0	0
x	t_{34}	x	0	x	0	x	0

So, when the 4×130 multiplication algorithm is applied during this case, we obtain the following output:

t_{30}	0	0	0

t_{31}	0	0	0

t_{32}	0	0	0

t_{33}	0	0	0

t_{34}	0	0	0

This result hints that some optimization is possible because of the following reasons:

- In case 1, the rightmost 6 32-bit blocks of the operand registers contain zeros.
- The lower order 192 bits of each of the 256-bit registers containing the product are zero.
- We need 25 multiplications only for a single message block and the other message blocks are set to zeros. Hence, SIMD implementation can be followed for this single message block.

If this concept is followed then the total number of *_mm256_mul_epu32* operations, which has latency 5, is reduced to 7. We add 48 zeros at the beginning of the message. But this addition of zeros is logical. No physical addition of zeros are required. The changed alignment is as follows:

0	t_{33}	0	t_{32}	0	t_{31}	0	t_{30}

This change in alignment is required only for those inputs which have a remaining block of length at least 1 byte and at most 16 bytes. No change in alignment is required for those inputs whose lengths satisfy one of the following conditions:

1. $17 \leq$ (`remaining length in bytes after 4-decimation Horner`) ≤ 31
2. $33 \leq$ (`remaining length in bytes after 4-decimation Horner`) ≤ 63

Modified Multiplication algorithm for case 1. Note: This change in alignment for inputs is required only for case 1 and noticeable speed-up has been obtained against [9] due to this changed alignment.

`Input:`

0	t_{33}	0	t_{32}	0	t_{31}	0	t_{30}
x	t_{34}	x	0	x	0	x	0

R^4 and $5 \cdot R^4$ aligned in 2 256-bit registers. All the alignments are done as discussed earlier.
Step 1:
To obtain partial results of t_{30}, t_{31}, t_{32} and t_{33}.
Operation involved: *_mm256_mul_epu32*

operand 1:	0	t_{33}	0	t_{32}	0	t_{31}	0	t_{30}
operand 2:	g_0	g_0	g_0	g_0	g_0	g_0	g_0	g_0
Product:	$t_{33} \cdot g_0$	$t_{32} \cdot g_0$	$t_{31} \cdot g_0$	$t_{30} \cdot g_0$				

Step 2:
To obtain partial results of t_{30}, t_{31}, t_{32} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	0	t_{32}	0	t_{31}	0	t_{30}	0	t_{33}
operand 2:	g_1	g_1	g_1	g_1	g_1	g_1	$5 \cdot g_2$	$5 \cdot g_2$
Product:	$t_{32} \cdot g_1$	$t_{31} \cdot g_1$	$t_{30} \cdot g_1$	$5 \cdot t_{33} \cdot g_2$				

Step 3:
To obtain partial results of t_{30}, t_{31}, t_{32} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	0	t_{31}	0	t_{30}	0	t_{33}	0	t_{32}
operand 2:	g_2	g_2	g_2	g_2	g_2	g_2	$5 \cdot g_3$	$5 \cdot g_3$
Product:	$t_{31} \cdot g_2$	$g_2 \cdot t_{30}$	$5 \cdot g_3 \cdot t_{33}$	$5 \cdot g_3 \cdot t_{32}$				

Step 4:
To obtain partial results of t_{30}, t_{31}, t_{32} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	0	t_{30}	0	t_{33}	0	t_{32}	0	t_{31}
operand 2:	g_3	g_3	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_4$
Product:	$t_{30} \cdot g_3$	$5 \cdot g_4 \cdot t_{33}$	$5 \cdot g_4 \cdot t_{32}$	$5 \cdot g_4 \cdot t_{31}$				

Step 5:
To obtain partial results of t_{30}, t_{31}, t_{32} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	0	t_{34}	0	t_{34}	0	t_{34}	0	t_{34}
operand 2:	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_3$	$5 \cdot g_3$	$5 \cdot g_2$	$5 \cdot g_2$	$5 \cdot g_1$	$5 \cdot g_1$
Product:	$5 \cdot g_4 \cdot t_{34}$	$5 \cdot g_3 \cdot t_{34}$	$5 \cdot g_2 \cdot t_{34}$	$5 \cdot g_1 \cdot t_{34}$				

Step 4:
To obtain partial results of t_{30}, t_{31}, t_{32} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	0	t_{30}	0	t_{33}	0	t_{32}	0	t_{31}

operand 2:	g_3	g_3	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_4$	$5 \cdot g_4$

Product:	$t_{30} \cdot g_3$	$5 \cdot g_4 \cdot t_{33}$	$5 \cdot g_4 \cdot t_{32}$	$5 \cdot g_4 \cdot t_{31}$

Step 6:
To obtain partial results of t_{30}, t_{31}, t_{32} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	0	t_{33}	0	t_{32}	0	t_{31}	0	t_{30}

operand 2:	g_1	g_1	g_2	g_2	g_3	g_3	g_4	g_4

Product:	$t_{33} \cdot g_1$	$t_{32} \cdot g_2$	$t_{31} \cdot g_3$	$t_{30} \cdot g_4$

Step 7:
To obtain partial results of t_{30}, t_{31}, t_{32} and t_{33}.
Operation involved: $\_mm256\_mul\_epu32$

operand 1:	0	t_{33}	0	t_{32}	0	t_{31}	0	t_{30}

operand 2:	g_1	g_1	g_2	g_2	g_3	g_3	g_4	g_4

Product:	$t_{33} \cdot g_1$	$t_{32} \cdot g_2$	$t_{31} \cdot g_3$	$t_{30} \cdot g_4$

Operation Count for the modified multiplication algorithm

Name of Intrinsic	Count
_mm256_mul_epu32	7
_mm256_set_epi64x	8
_mm256_add_epi64	7
_mm256_permutevar8x32_epi32	9
_mm256_permute4x64_epi64	4
_mm256_unpacklo_epi64	1
_mm256_unpackhi_epi64	1
_mm256_blend_epi32	1

5 Result Analysis

5.1 Environment for Measuring Machine Cycles.

Performance has been measured in terms of number of machine cycles per byte using the same conditions as mentioned in [8]. Intel Turbo Boost Technology, Intel Hyper-Threading Technology and Intel Speedstep Technology were disabled. The following code was used for measuring time using appropiate number of iterations for cache warming and stabilizing the cycles/byte result.

```
int COMP(const void*a,const void *b) return ( *(double*)a - *(double*)b );
#define MEASURE(f)
MIN_RESULT = 1.7976931348623158e+308;
for(R_CNT=0;R_CNT < 5;R_CNT++)
for(W_CNT=0; W_CNT < WARMUP_REPS; W_CNT++)
f
for(M_CNT=0; M_CNT < MEASURE_REPS; M_CNT++)
START = getticks();
f
END = getticks();
RESULT = (double)(END - START);
time_record[M_CNT]=RESULT;


qsort(time_record,MEASURE_REPS,sizeof(double),COMP);
if(MEASURE_REPS%2)
MEDIAN=time_record[MEASURE_REPS/2];
else
MEDIAN=(time_record[MEASURE_REPS/2]+time_record[(MEASURE_REPS/2)+1])/2;
if(MEDIAN<MIN_RESULT)
MIN_RESULT=MEDIAN;
```

5.2 Results for Skylake and Kaby Lake

Specifications for Skylake

- Software details
  - Operating System: Ubuntu 14.04 LTS (64-bit)
  - Compiler: gcc version 5.5.0 with AVX2 support ('-mavx2' flag has been used) and compile time optimizations like '-O3' and '-fomit-frame-pointer' were used.
- Hardware details:
  - Intel Core i7-6500U CPU @ 2.50GHz $\times$ 2 Skylake GT2.

Specifications for Kaby Lake

- Software details
  - Operating System: Ubuntu 18.04 LTS (64-bit)
  - Compiler: gcc version 7.3.0 with AVX2 support ('-mavx2' flag has been used) and compile time optimizations like '-O3' and '-fomit-frame-pointer' were used.
- Hardware details:
  - Intel Core i7-7700U CPU @ 3.60GHz $\times$ 4 Kaby Lake GT2.

This modified implementation aimed at improving performance for small messages (up to length 4KB). Similar kind of performance has been observed both in Skylake and Kaby Lake cores. The result analysis is as follows.

Significant speed-up has been obtained for messages satisfying case 3 as discussed in Section 3.3 roughly up to 1KB length. Beyond this, noticeable improvement has been observed. Similar kind of result has been obtained for cases where $49 \leq n' \leq 63,\ 33 \leq n' \leq 47$ and $17 \leq n' \leq 31$.
Noticeable speed-up has been obtained in significant number of cases for messages satisfying Case 2 and Case 1 for Section 3.3. Same kind of result has been obtained for cases where $1 \leq n' \leq 15$.

This version gives speed-up for messages till length 4KB. A very brief summary of the performance for 256-bit vectorization using AVX2 is as follows:

- For Skylake core speed-up has been obtained in 93.36% cases. For 5.31% cases, this version gives same performance as that of [9]. We have not been able to gain speed-up in 1.33% cases.
- For Kaby Lake core speed-up has been obtained in 92.60% cases. For 6.19% cases, this version gives same performance as that of [9]. We have not been able to gain speed-up in 1.21% cases.
- We have obtained maximum 34.46% speed-up and an average of 12.58% speed-up till 1 KB size in Skylake processor and maximum 34.63% speed-up and an average of 14.44% speed-up till 1 KB size for Kaby Lake processor.
- For message size between 1 KB and 2 KB maximum speed-up of 20.69% with average 6.10% in Skylake core and for Kaby Lake Core maximum speed-up is 16.03% with an average speed-up of 6.53% have been obtained.
- For message size between 2 KB and 3 KB maximum speed-up of 13.79% with average 3.92% in Skylake core and for Kaby Lake Core maximum speed-up is 13.79% with an average speed-up of 4.13% have been obtained.
- For message size between 3 KB and 4 KB maximum speed-up of 10.26% with average 2.90% in Skylake core and for Kaby Lake Core maximum speed-up is 13.75% with an average speed-up of 2.64% have been obtained.

The cases where more number of fresh packings of operands are required higher speed-up is obtained compared to those cases where the number of fresh packings are less.

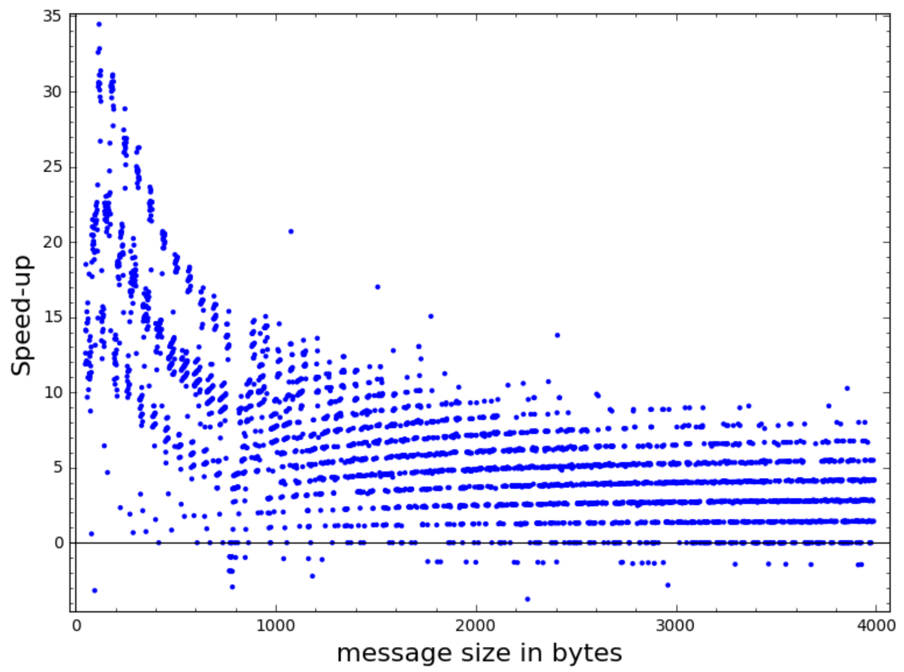

Figure 1: Speed-up vs message size in bytes graph for Skylake core

Figure 1 shows the speed-up vs message size in bytes graph for Skylake core. Figure 8 shows the speed-up vs message size in bytes graph for Kaby Lake core. Figure 2, Figure 3, Figure 4, Figure 5, Figure 6, Figure 7 show the cycles/byte vs message size in bytes graphs for the indicated message-length ranges for Skylake core. Figure 9, Figure 10, Figure 11, Figure 12, Figure 13, Figure 14 for Kaby Lake core. All the experiments have been done on message lengths ranging from 49B to 4000KB except the ones having size in integral powers of 2.

Table 1 and Table 2 give some raw data obtained from experiments for Skylake core and Kaby Lake core respectively.

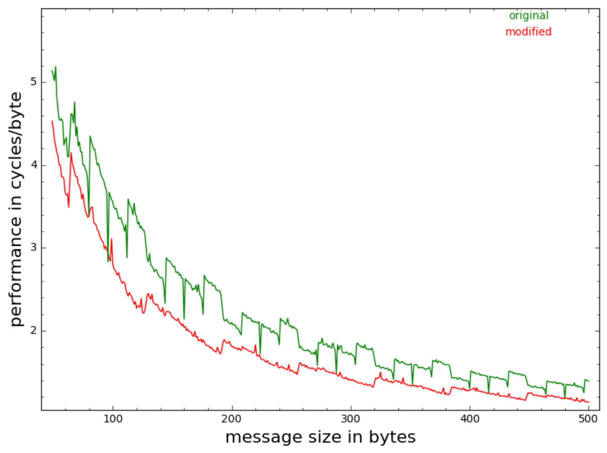

Figure 2: cycles/byte vs message size in bytes (49 - 500 bytes) graph for Skylake core

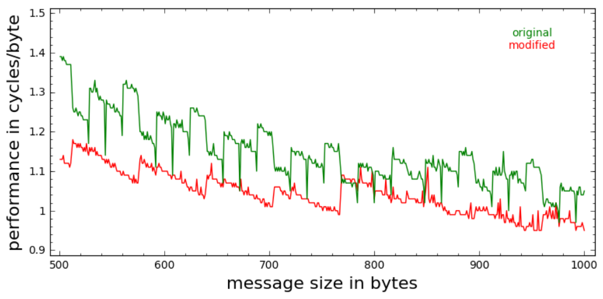

Figure 3: cycles/byte vs message size in bytes (501 - 1000 bytes) graph for Skylake core

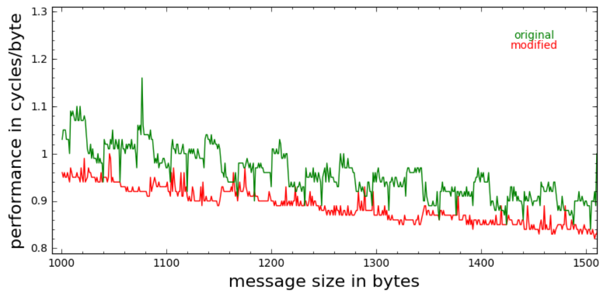

Figure 4: cycles/byte vs message size in bytes (1001 - 1500 bytes) graph for Skylake core

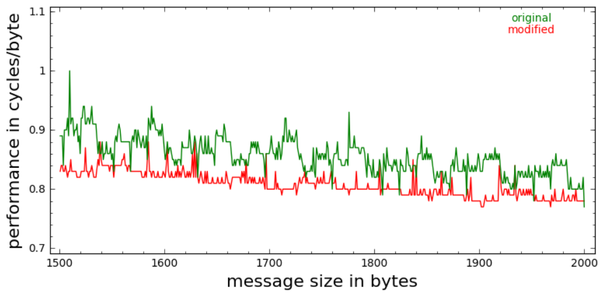

Figure 5: cycles/byte vs message size in bytes (1501 - 2000 bytes) graph for Skylake core

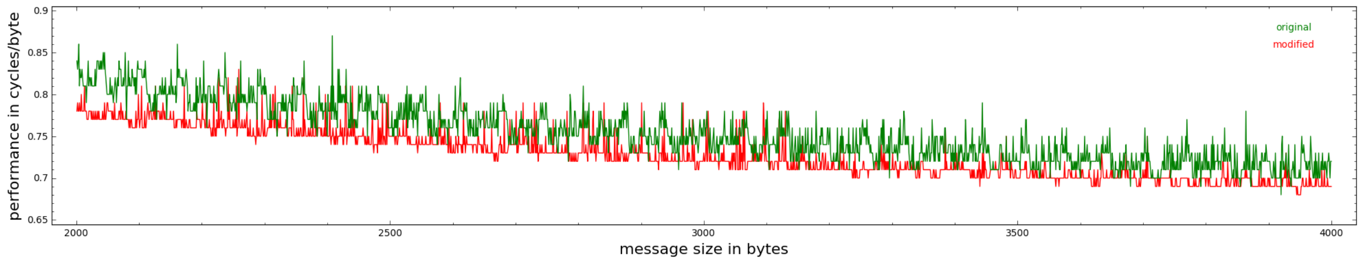

Figure 6: cycles/byte vs message size in bytes (2001 - 4000 bytes) graph for Skylake core

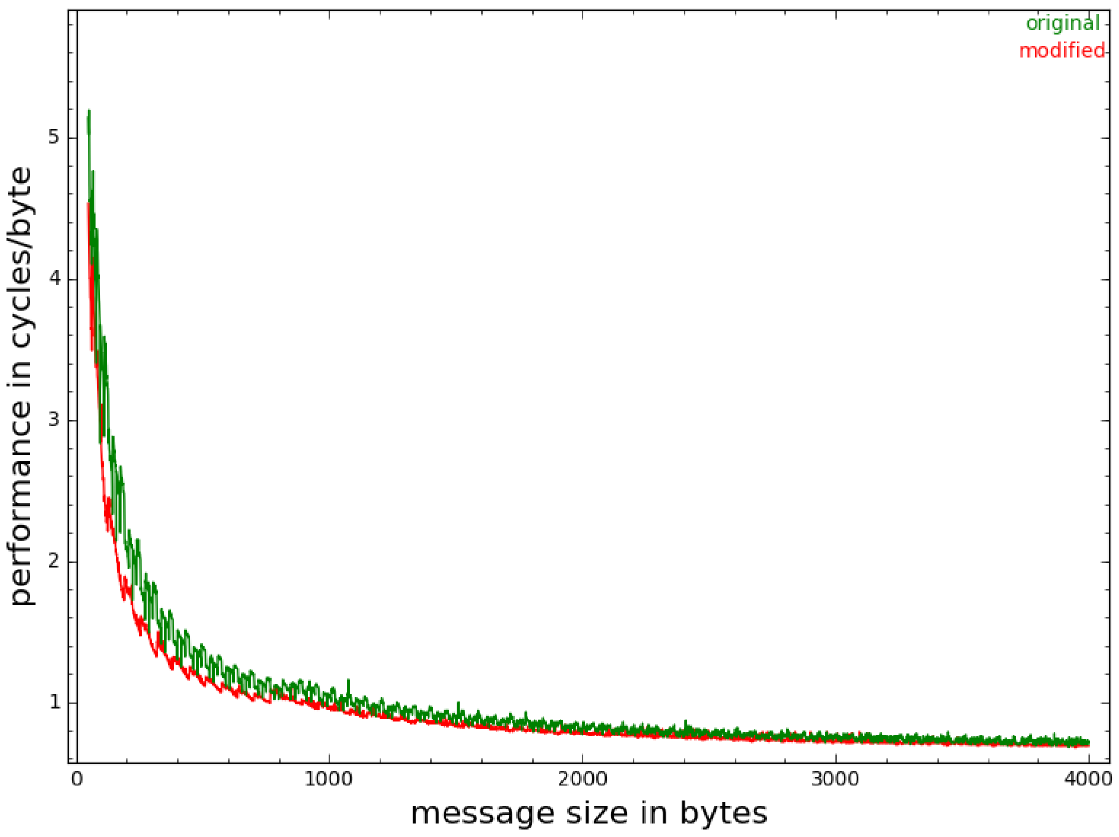

Figure 7: cycles/byte vs message size in bytes (49 - 4000 bytes) graph for Skylake core

The following table gives timing data for some selected message length. The claimed nature of speed-up for Skylake core can be verified from this table and the figures given earlier.

Table 1: Some results as observed for a Skylake core

Message length in bytes	cycles/byte for original	cycles/byte for modified	speed-up
65	4.62	4.15	10.17
66	4.61	4.03	12.58
67	4.51	3.97	11.97
68	4.76	3.91	17.86
69	4.35	3.86	11.26
70	4.46	3.86	13.45
71	4.23	3.77	10.87
72	4.28	3.75	12.38
73	4.16	3.7	11.06
74	4.16	3.59	13.70
75	4.00	3.65	8.75
76	4.00	3.53	11.75
77	3.95	3.45	12.66
78	3.92	3.41	13.01
79	3.80	3.37	11.32
80	3.40	3.38	0.59
81	4.35	3.46	20.46
95	3.86	2.97	19.29
96	2.83	2.92	-3.18
97	3.67	2.87	21.80
111	3.28	2.5	23.78
112	2.88	2.45	14.93
113	3.59	2.42	32.59
114	3.53	2.46	30.31
115	3.50	2.43	30.57
116	3.43	2.41	30.55
117	3.40	2.36	30.59
118	3.54	2.82	34.46
119	3.41	2.35	31.09
120	3.38	2.27	32.84
121	3.29	2.30	30.09
122	3.31	2.30	30.51
123	3.24	2.28	29.63
124	3.26	2.39	26.69
125	3.22	2.22	31.06
126	3.22	2.21	31.37
127	3.17	2.24	29.34
569	1.32	1.08	18.18
570	1.31	1.07	18.32

575	1.29	1.07	17.05
577	1.20	1.13	5.83
592	1.11	1.09	1.80
593	1.25	1.11	11.20
594	1.24	1.11	10.48
595	1.25	1.12	10.40
596	1.24	1.12	10.48
612	1.23	1.08	12.20
639	1.23	1.04	15.45
1473	0.88	0.85	3.41
1474	0.89	0.85	4.49
1475	0.88	0.85	3.41
1476	0.88	0.85	3.41
1477	0.91	0.84	7.69
1478	0.89	0.85	4.49
1479	0.88	0.85	3.41
1480	0.88	0.84	4.55
1481	0.88	0.85	3.41
1482	0.88	0.84	4.55
1483	0.88	0.84	4.55
1484	0.87	0.84	3.45
1485	0.87	0.84	3.45
1486	0.88	0.86	2.27
1487	0.87	0.85	2.30
1488	0.85	0.84	1.18
1500	0.90	0.84	6.67
1529	0.91	0.83	8.79
1530	0.92	0.83	9.78
1531	0.94	0.84	10.64
1532	0.91	0.83	8.79
1533	0.91	0.82	9.89

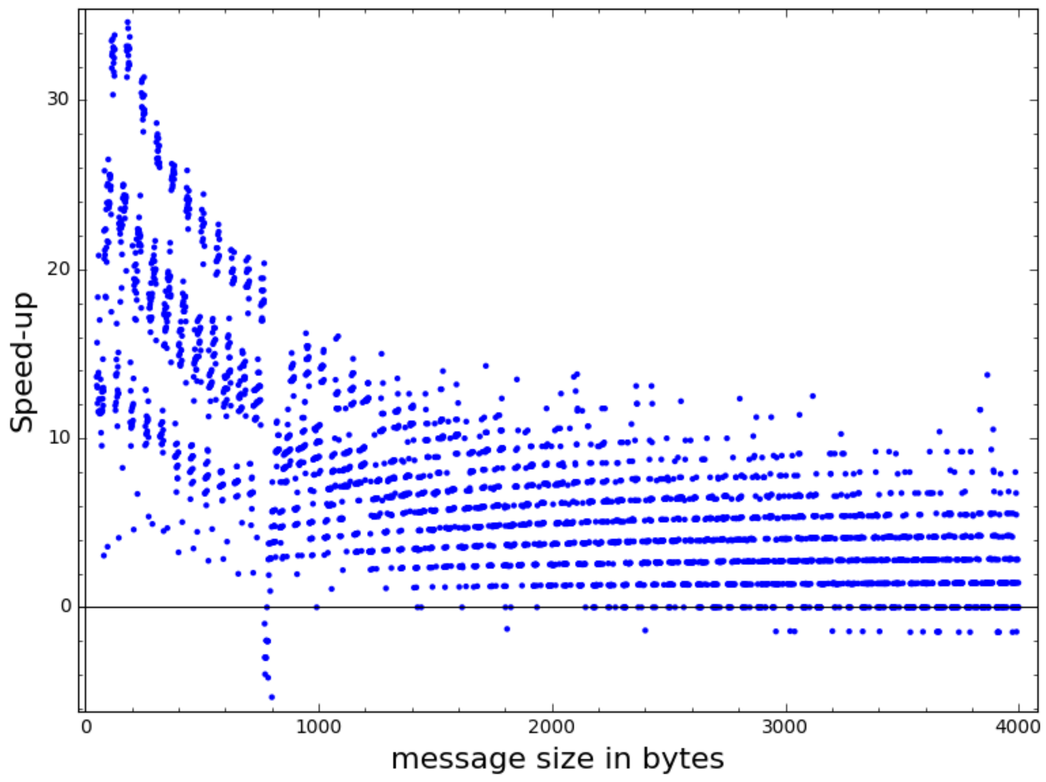

Figure 8: Speed-up vs message size in bytes graph for Kaby Lake core

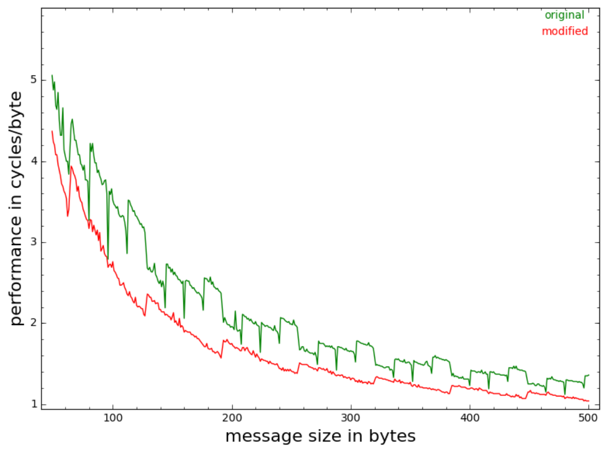

Figure 9: cycles/byte vs message size in bytes (49 - 500 bytes) graph for Kaby Lake core

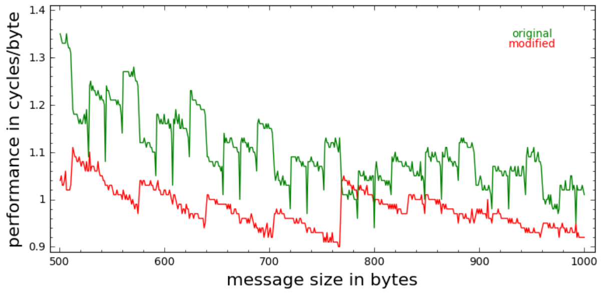

Figure 10: cycles/byte vs message size in bytes (501 - 1000 bytes) graph for Kaby Lake core

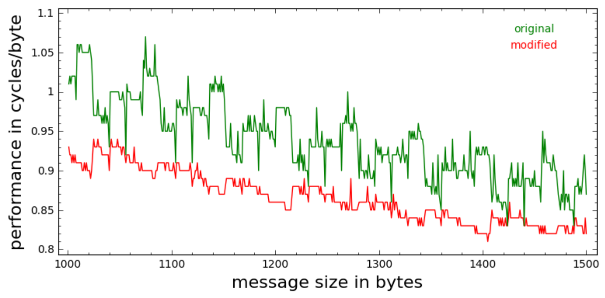

Figure 11: cycles/byte vs message size in bytes (1001 - 1500 bytes) graph for Kaby Lake core

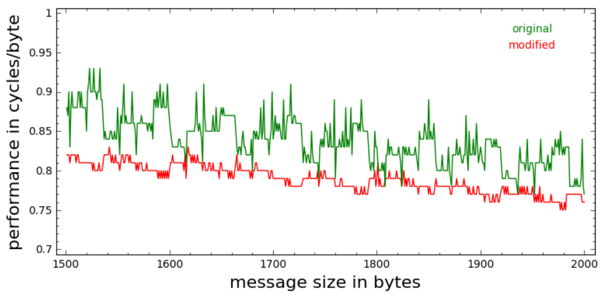

Figure 12: cycles/byte vs message size in bytes (1501 - 2000 bytes) graph for Kaby Lake core

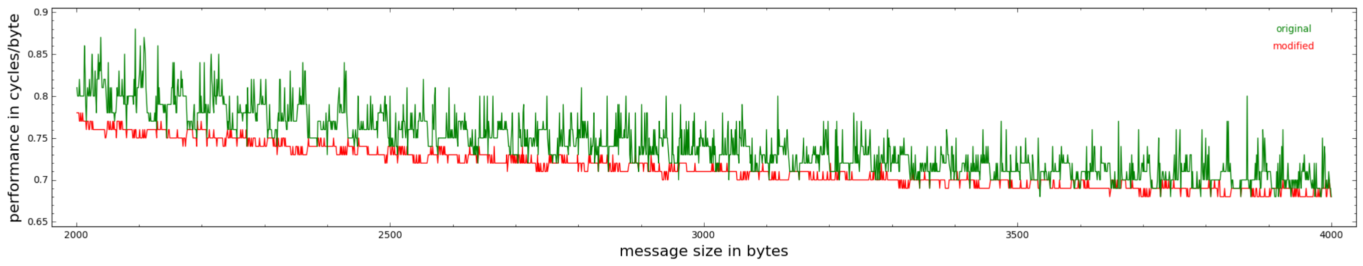

Figure 13: cycles/byte vs message size in bytes (2001 - 4000 bytes) graph for Kaby Lake core

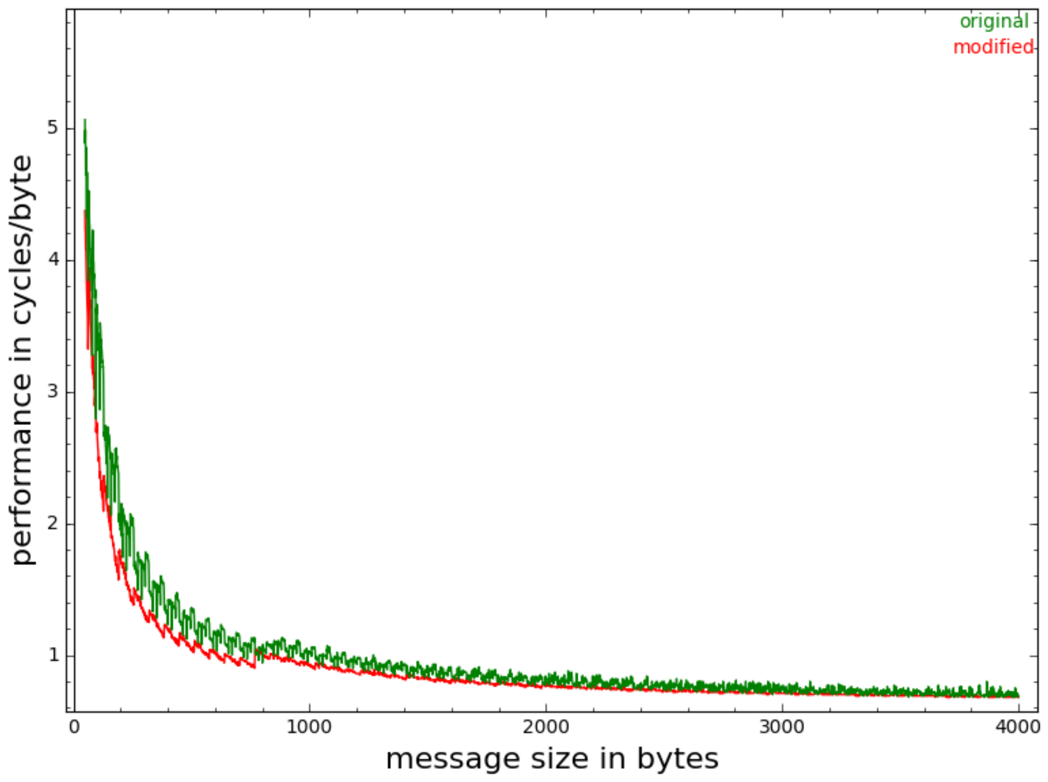

Figure 14: cycles/byte vs message size in bytes (49 - 4000 bytes) graph for Kaby Lake core

The following table gives timing data for some selected message length. The claimed nature of speed-up for Kaby Lake core can be verified from this table and the figures given earlier.

Table 2: Some results as observed for a Kaby Lake core

Message length in bytes	cycles/byte for original	cycles/byte for modified	speed-up
65	4.46	3.94	11.65
66	4.52	3.91	13.49
67	4.39	3.85	12.30
68	4.26	3.82	10.32
69	4.26	3.77	11.50
70	4.17	3.63	12.95
71	4.08	3.69	9.56
72	4.08	3.56	12.74
73	3.97	3.51	11.59
74	3.95	3.49	11.64
75	3.89	3.41	12.34
76	3.95	3.37	14.68
77	3.77	3.32	11.94
78	3.77	3.28	13.00
79	3.75	3.27	12.80

80	3.27	3.17	3.06
81	4.22	3.28	22.27
95	3.60	2.82	21.67
96	2.79	2.69	3.58
97	3.63	2.72	25.07
111	3.14	2.41	23.25
112	2.86	2.36	17.48
113	3.52	2.34	33.52
114	3.51	2.39	31.91
115	3.46	2.33	32.66
116	3.43	2.31	32.65
117	3.38	2.27	32.84
118	3.39	2.25	33.63
119	3.33	2.32	30.33
120	3.32	2.22	33.13
121	3.29	2.20	33.13
122	3.26	2.21	32.21
123	3.22	2.20	31.68
124	3.23	2.18	32.51
125	3.18	2.18	31.45
126	3.19	2.11	33.86
127	3.12	2.09	33.01
569	1.27	0.99	22.05
570	1.26	1.00	20.63
575	1.24	0.97	21.77
577	1.12	1.04	7.14
592	1.05	1.02	2.86
593	1.18	1.03	12.71
594	1.18	1.04	11.86
595	1.17	1.02	12.82
596	1.16	1.02	12.07
612	1.17	0.99	15.38
639	1.18	0.95	19.49
1473	0.87	0.83	4.60
1474	0.86	0.83	3.49
1475	0.85	0.83	2.35
1476	0.87	0.83	4.60
1477	0.86	0.83	3.49
1478	0.90	0.83	7.78
1479	0.85	0.83	2.35
1480	0.87	0.82	5.75
1481	0.87	0.83	4.60
1482	0.86	0.83	3.49
1483	0.87	0.82	5.75

1484	0.85	0.82	3.53
1485	0.85	0.82	3.53
1486	0.85	0.82	3.53
1487	0.85	0.83	2.35
1488	0.83	0.82	1.20

6 512-bit vectorization of Poly1305

In [8], the 256-bit vectorization has also been extended to 512-bit vectorization. In this case 8-decimation Horner has been used for messages where the number of 16-byte blocks is a multiple of 8. Otherwise, incomplete 8-decimation has been used in the same way as 4-decimation Horner has been used incompletely for 256-bit vectorization. The message space has been partitioned is a similar way. For messages whose number of 16-byte blocks is not a multiple of 8, repeated fresh packing are required. Hence, again we have used this gap for improvement.

8-decimation Horner. Given a sequence of ℓ 130-bit blocks $M_0, ..., M_{\ell-1}$, $\mathbf{Horner}_R\left(M_0, ..., M_{\ell-1}\right)$ can be computed in the following way which is termed as 8-decimation Horner [7].

$$\begin{aligned}
\mathbf{Horner}_R\left(M_0, ..., M_{\ell-1}\right) \;=\; & R^8\mathbf{Horner}_{R^8}\left(M_0, M_8, ..., M_{8i}, ...\right) \\
& +R^7\mathbf{Horner}_{R^8}\left(M_1, M_9, ..., M_{8i+1}, ...\right) \\
& +R^6\mathbf{Horner}_{R^8}\left(M_2, M_{10}, ..., M_{8i+2}, ...\right) \\
& +R^5\mathbf{Horner}_{R^8}\left(M_3, M_{11}, ..., M_{8i+3}, ...\right) \\
& +R^4\mathbf{Horner}_{R^8}\left(M_4, M_{12}, ..., M_{8i+4}, ...\right) \\
& +R^3\mathbf{Horner}_{R^8}\left(M_5, M_{13}, ..., M_{8i+5}, ...\right) \\
& +R^2\mathbf{Horner}_{R^8}\left(M_6, M_{14}, ..., M_{8i+6}, ...\right) \\
& +R\mathbf{Horner}_{R^8}\left(M_7, M_{15}, ..., M_{8i+7}, ...\right)
\end{aligned}$$

Partitioning The Message Space. We have used variants of 8-decimation Horner for improvement. In the improved version of 512-bit vectorization of Poly1305, we follow the a variant of 8-decimation Horner for computation irrespective of the message length and thus the message space can be partitioned into the following 8 disjoint cases.

- case 0: messages where number of 16-byte blocks $\equiv 0 \bmod 8$
- case 1: messages where number of 16-byte blocks $\equiv 1 \bmod 8$
- case 2: messages where number of 16-byte blocks $\equiv 2 \bmod 8$
- case 3: messages where number of 16-byte blocks $\equiv 3 \bmod 8$
- case 4: messages where number of 16-byte blocks $\equiv 4 \bmod 8$
- case 5: messages where number of 16-byte blocks $\equiv 5 \bmod 8$

- case 6: messages where number of 16-byte blocks $\equiv 6 \bmod 8$
- case 7: messages where number of 16-byte blocks $\equiv 7 \bmod 8$

Based on the above mentioned cases, we have used variants of 8-decimation Horner for evaluation.

Intel's most recent processors like Skylake-X, Cannonlake and Ice Lake provide AVX-512 extension. Since no such processor is available to us, we cannot find the cycles/byte for modified 512-bit vectorized version. We have compiled the code using gcc 5.5.0 and the resulting binary has been executed using the Software Development Emulator [3]. The instruction count obtained from the modified version is lesser than the Goll-Gueron version. But the detailed result analysis is not available due to time constraint.

7 Conclusion

Horner's rule is used widely for evaluation of polynomial-based hash. Intel intrinsics, together with the underlying microarchitecture, allows parallelization of such evaluation. Hence, this work not only gives improved SIMD implementation of Poly1305, our concept of balancing can be used in a wider scope. From tables and figures (for 256-bit vectorization) given in the earlier section it is clear that this modification gives considerable speed-up for Poly1305 which is an important polynomial-based hash function. For small messages ($\leq$1 KB) significant speed-up has been obtained. Noticeable speed-up has been obtained till 4KB message size for modern Intel microarchitectures. So this work has practical importance.

References

[1] Google Security Blog *https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html*

[2] Intel Intrinsics Guide *https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX,AVX2*

[3] Intel: *Intel Software Development Emulator (SDE)* Available from http://software.intel.com/en-us/articles/intel-software-development- emulator

[4] Wikipedia *https://en.wikipedia.org/wiki/Poly1305*

[5] Daniel J.Bernstein *The Poly1305-AES message-authentication code* In Fast Software Encryption: 12th international workshop, FSE 2005, pages 32-49, March 2005, http://cr.yp.to/mac/poly1305-20050329.pdf

[6] Daniel J. Bernstein and Peter Schwabe *NEON crypto* In Cryptographic hardware and embedded systems - CHES 2012. 14th international workshop, Leuven, Belgium, September 912, 2012.

[7] Debrup Chakraborty, Sebati Ghosh and Palash Sarkar *A Fast Single-Key Two-Level Universal Hash Function.* IACR Trans. Symmetric Cryptol. 2017(1): 106-128 (2017)

[8] Martin Goll, Shay Gueron *Vectorization of Poly1305 Message Authentication Code.* In 12th International Conference on Information Technology-New Generation, 2015

[9] Martin Goll, Shay Gueron *256-bit vectorized implementation of Poly1305 using AVX2* The code was obtained from Shay Gueron. Also it has been made public as OpenSSL patch.