Improving the performance of Luffa Hash Algorithm

Thomaz Oliveira[1] **, Julio López**[1]

August 19, 2010

[1]Instituto de Computação, Universidade Estadual de Campinas

thomaz.oliveira@students.ic.unicamp.br, jlopez@ic.unicamp.br

Abstract. *Luffa is a new hash algorithm that has been accepted for round two of the NIST hash function competition SHA-3. Computational efficiency is the second most important evaluation criteria used to compare candidate algorithms. In this paper, we describe a fast software implementation of the Luffa hash algorithm for the Intel Core 2 Duo platform. We explore the use of the perfect shuffle operation to improve the performance of 64-bit implementation and 128-bit implementation with the Intel Supplemental SSSE3 instructions. In addition, we introduce a new way of implementing Luffa based on a Parallel Table Lookup instruction. The timings of our 64-bit implementation (C code) resulted in a 16 to 32% speed improvement over the previous fastest implementation.*

1. Introduction

Hash functions are an important component of many security applications, such as digital signature schemes, data identification and key derivation. Its main use is to provide data integrity and message authentication. One of the most used hash algorithm is the SHA-1 [NIST 2002], published by NIST (National Institute of Standards and Technology) and adopted by important security protocols, like IPsec, TLS/SSL and SSH.

Recent attacks on the widely used SHA-1 [Wang et al. 2005, Biham et al. 2005] compromised its security. This scenario motivated NIST to launch a public competition for the selection of a new secure hash function, the SHA-3. The competition counted initially with 64 submissions.

For the second round, NIST selected 14 candidates using the following three categories of evaluation criteria: 1) security, 2) cost and performance and 3) algorithm and implementation characteristics. The Intel Core 2 Duo processor was chosen as a reference to evaluate the software performance of the candidates.

Luffa is a family of four hash functions: Luffa-224, Luffa-256, Luffa-384 and Luffa-512 [Canniere et al. 2009a], designed by Carrière, Sato and Watanabe. In December 2009 the Luffa hash algorithm was selected for the second round. Its design is based on a variant of the sponge construction [Bertoni et al. 2007], processing multiple permutations in parallel.

Since its submission, there have been some implementations of Luffa, both in hardware and software. The authors provided software implementations for 8, 32 and 64-bit platforms [Canniere et al. 2009b], exploring the algorithm inherent parallelism through SSE (Streaming SIMD Extensions). They also submitted implementations for the eBACS project [Bernstein and Lange]. In [Pornin 2010] Pornin reported a software

implementation of Luffa in C and Java, and included in his sphlib library [Pornin]. The authors in [Oikawa et al. 2010] provided implementations for GPUs (Graphics Processing Unit), processing lots of different messages. For hardware implementation, recent works [Kobayashi et al. 2010, Tillich et al. 2009, Knežević and Verbauwhede 2009, Namin and Hasan 2009] show that Luffa ranked among the fastest algorithms in comparison to other SHA-3 second round candidates.

In this paper, we focus on software implementation of Luffa hash algorithm for the Intel Core 2 Duo processor. Our work takes advantage of the parallelism provided by the algorithm and combines it to a technique called perfect shuffle [Warren 2002] and the SSE vector instructions. The performance results of our optimized C code for 64-bit implementation outperforms the Reference Implementation version 2.01 [Canniere et al. 2009a]. In addition, we describe a new approach to implement Luffa for 128-bit registers.

Organization of the paper A brief description of Luffa algorithm is presented in Section 2. In Section 3 we explain the perfect shuffle technique. Section 4 describes the SSE shuffle instructions and their usage. The application of the techniques in our implementation is presented in Section 5. Finally, Section 6 gives the experimental results for Intel Core 2 Duo processor with different compilers.

2. The Luffa algorithm

Luffa operates by iterating a mixing function called round function; its final transformation consists of iterations of an output function and a round function with a fixed message 0^{256}. The round function has fixed length input and output and differ according to the message digest size. It takes as input a message block and a chaining variable.

Table 1. Luffa attributes

Hash length (bits)	**Message block size** (bits)	**Chaining size** (bits)	**Number of permutations**
224	256	3 blocks of 256 (768)	3
256	256	3 blocks of 256 (768)	3
384	256	4 blocks of 256 (1024)	4
512	256	5 blocks of 256 (1280)	5

The round function is divided in two parts: the Message Injection and the nonlinear permutations. In this Section we will focus on the permutations due to its significance to our work. Technical details of Luffa can be found in [Canniere et al. 2009a, Canniere et al. 2009b].

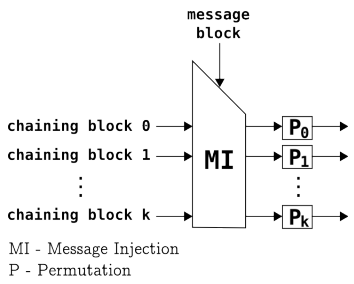

Figure 1. Luffa round function

2.1. Message Injection

In this phase, the message block is mixed with the chaining variable; this mixing is made through functions represented by matrices over a ring $GF((2^{32})^8)$. For example, the message injection function for Luffa-224 and Luffa-256 is presented below:

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} 3 & 2 & 2 & 1 \\ 2 & 3 & 2 & 2 \\ 2 & 2 & 3 & 4 \end{pmatrix} \begin{pmatrix} H_0 \\ H_1 \\ H_2 \\ M \end{pmatrix}$$

In [Canniere et al. 2009b], the authors suggest the implementation of the message injection functions using only multiplications by x (0x02) and XOR instructions. Therefore, the function presented above can be implemented through scheme depicted in Figure 2.

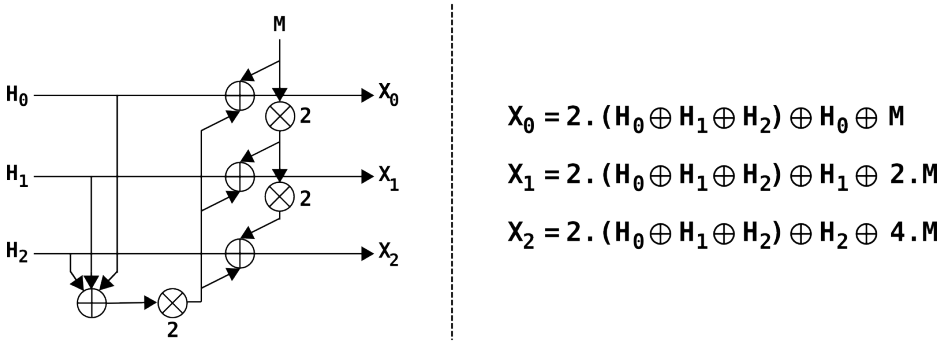

Figure 2. Message Injection function for Luffa-224 and Luffa-256

2.2. Permutation

The permutations have input and output size of 256 bits, and their number will vary with the message digest size, as shown in Table 1. Each permutation consists of a Tweak and eight iterations of the following functions, also called the step functions: SubCrumb, MixWord and AddConstant.

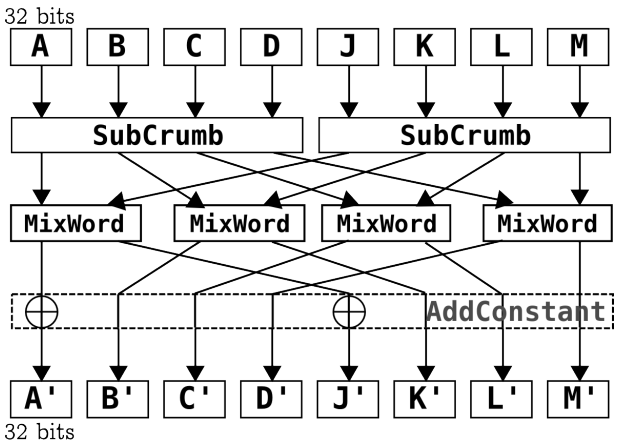

Figure 3. Luffa step functions

2.2.1. Tweak

This step is processed only once for each permutation, and it is defined as a n-bit left rotation at the last four 32-bit words of a 256-bit input. The n value is defined according to the chaining block number.

2.2.2. SubCrumb

SubCrumb is a bit slice substitution function; it takes four bits in the same position from four 32-bit words and uses as input to a S-box defined below.

$$S[16] = \{13, 14, 0, 1, 5, 10, 7, 6, 11, 3, 9, 12, 15, 8, 2, 4\}$$

Then, the resulting bits from S are returned to the four words in the same position. This operation is done for each of the 32 bits of the input words.

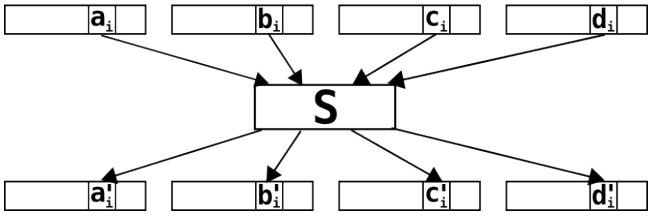

Figure 4. SubCrumb function applied to the i^{th} bits of the input words

In the Luffa specification, the authors suggest to implement the SubCrumb function using logical operations [Canniere et al. 2009a], whose sequences of instructions are shown in Table 2.

Table 2. SubCrumb instructions for Intel Core 2 Duo

MOV r4 r0	**OR** r0 r1	**XOR** r2 r3
NOT r1	**XOR** r0 r3	**AND** r3 r4
XOR r1 r3	**XOR** r3 r2	**AND** r2 r0
NOT r0	**XOR** r2 r1	**OR** r1 r3
XOR r4 r1	**XOR** r3 r2	**AND** r2 r1
XOR r1 r0		

2.2.3. MixWord

This function does a linear permutation of two 32-bit words through left rotations and XORs. The scheme is presented in Figure 5.

2.2.4. AddConstant

The Luffa constants are XORed to the chaining variable 32-bit words 0 and 4. Those constants are different for each chaining block number and for each iteration of the step functions; they consist of sixteen 32-bit words for each chaining block.

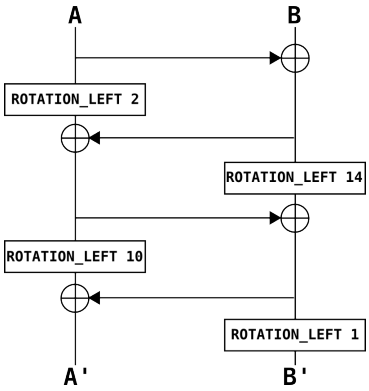

Figure 5. MixWord function

The algorithm must execute three to five permutations for each iteration, depending on the message digest size. However, the permutations are independent and can be executed simultaneously. This attribute will be explored through the 64-bit and 128-bit SSE registers and the instruction sets designed to work with them.

3. Perfect Shuffle

The perfect shuffle is a bit permutation that perfectly interleaves the bits of two or more words [Warren 2002]. This operation for two words is illustrated in Figure 6.

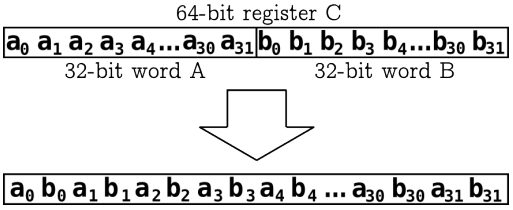

Figure 6. A perfect shuffle of two 32-bit words inside a 64-bit register

Some operations, such as the rotations, become simpler when the perfect shuffle operation is used. For example, to do a left rotation of the two 32-bit words A and B from Figure 6 by n bits without perfect shuffling we would need the following instructions on x86-64 architecture:

//rax holds the C register and rbx is an auxiliary register

mov rbx, rax
shl rax, \$$n$
shr rbx, \$(32-n)
and rax, MASK0 *//it is necessary to apply masks to clean the*
and rbx, MASK1 *//bits that have crossed the words boundaries.*
xor rax, rbx

Requiring six instructions. However, with the interleaved bits of two 32-bit words, a left rotation of n bits can be done with just one instruction:

//rax holds C register

rotl rax, $(2*$n$)

A similar operation, called bit permutation, is proposed in the Keccak algorithm [Bertoni et al. 2010] for speeding up the 32-bit software implementation.

4. Streaming SIMD Extensions

The SSE instruction set was designed by Intel and first released in Pentium III processors. To this first set more instructions were added, originating different versions: SSE2, SSE3, SSSE3, SSE4 [Intel 2002] and the AVX [Intel 2010]. SSE provides 128-bit registers, along with the 70 new instructions. One of the most powerful operation is the shuffle.

4.1. Shuffle instruction

The shuffle instruction requires two operands and a mask. The resulting register gets the elements from the operands according to the mask. As an example, a shuffle SSE C intrinsic [Intel 2002] applied to a SSE 128-bit register with four packed 32-bit words is shown in Figure 7 for swapping words B and D.

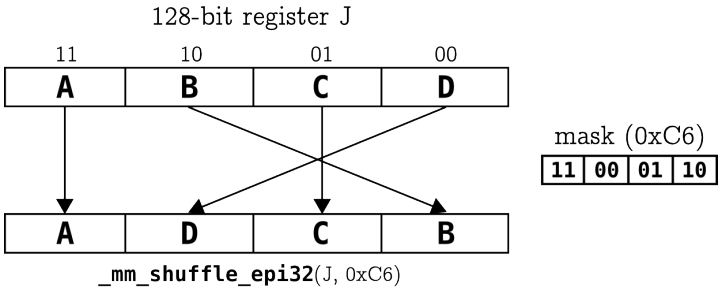

Figure 7. Shuffle instruction

Changing the mask allows the programmer to produce specific operations such as right or left rotations, as shown in Figure 8.

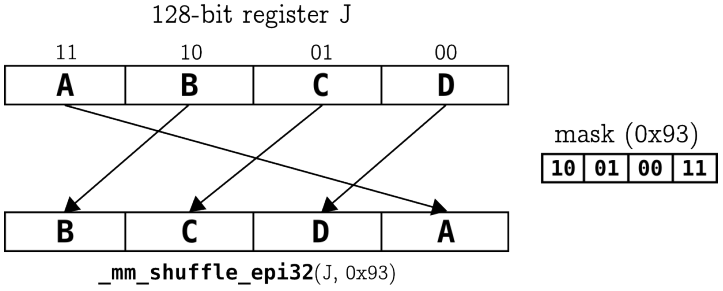

Figure 8. Left rotation through shuffle instruction

Byte shuffle has the same characteristic as the 32-bit shuffle instructions presented above, but the mask is now represented by another 128-bit register. The byte shuffle can also be used as a Parallel Table Lookup (PTLU) instruction. If the programmer uses the table as the operand and the operand as the mask, an 8-bit PTLU would be implemented. An example is depicted in Figure 9.

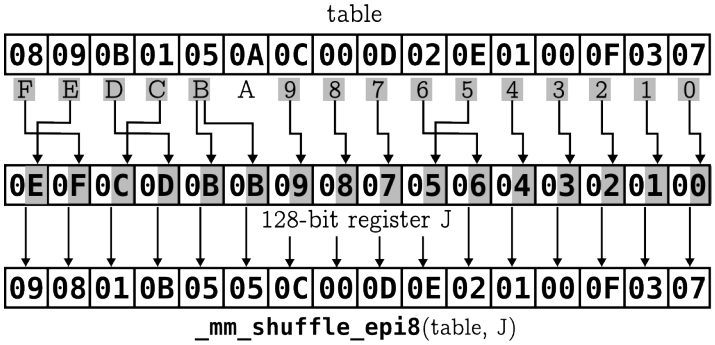

Figure 9. Byte shuffle implemented as a PTLU

5. Proposed Technique

In this section we explain how the operations presented in the previous sections were used to implement Luffa for 64-bit and SSE platforms.

5.1. 64-bit platform

In this scenario, data parallelism is possible through the allocation of two 32-bit words from different chaining blocks in one 64-bit register, as depicted in Figure 10. In this way, we can process two permutations simultaneously.

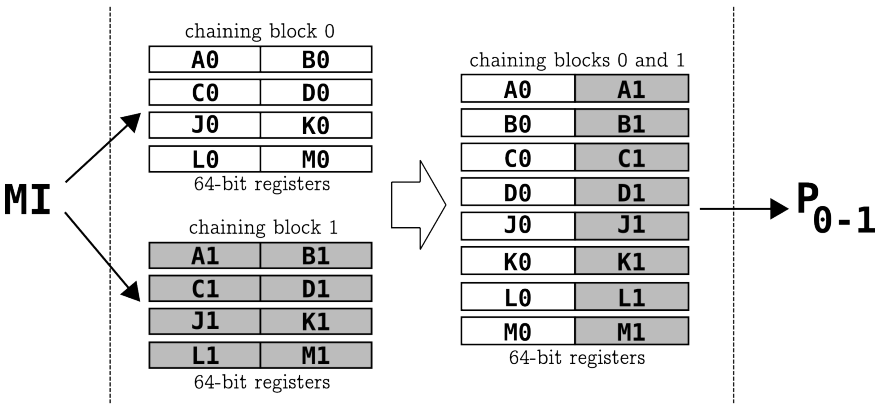

Figure 10. 64-bit parallelism

However, this way of disposing the two 32-bit word causes a huge overhead to the rotations processing, an operation often executed in Luffa, mainly in the MixWord. This function cost was increased by 139% in comparison to the authors' method (representing each 32-bit word in a distinct register). The reason was the number of instructions needed to rotate the two words, as shown in Section 3.

Through the application of perfect shuffle we could reduce the rotations cost to a single instruction. To keep the perfect shuffle overhead as low as possible, we apply the perfect shuffle only once per round function in the message block, maintaining this notation throughout the algorithm. The reverse transformation is done once, before computing the hash value. This scheme is shown in Figure 11.

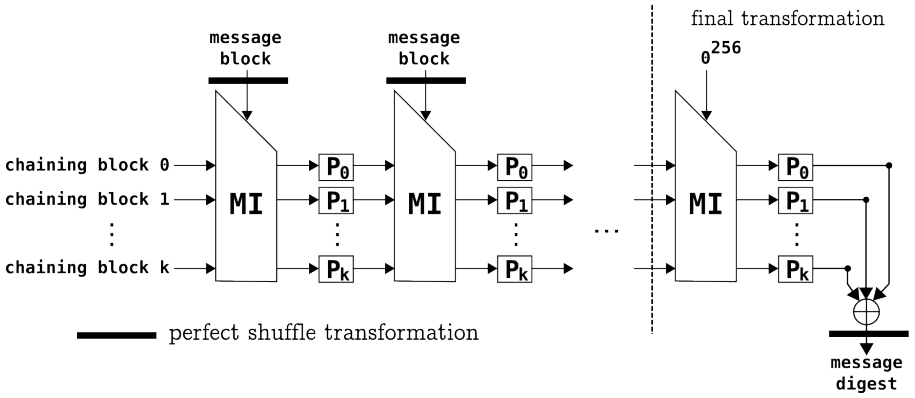

Figure 11. Perfect Shuffle in Luffa

The overhead of this implementation consists of two operations: disposing the registers as depicted in Figure 10, which costs about 2 cycles/byte; and the perfect shuffle applied on the message block, which costs about 3 cycles/byte using the implementation suggested in [Warren 2002]. Therefore, the total overhead cost resulted in about 5 cycles/byte.

Luffa-384 is the most benefited version, because it has to process four chaining block permutations, occupying completely two 64-bit registers. For this version of Luffa, the performance was improved by 32%.

5.2. SSE platform

In this scenario, we have 128-bit registers that can allocate four 32-bit words. As a consequence, four permutations can be processed in the same time. The authors' method to dispose the words inside the registers is depicted in Figure 12.

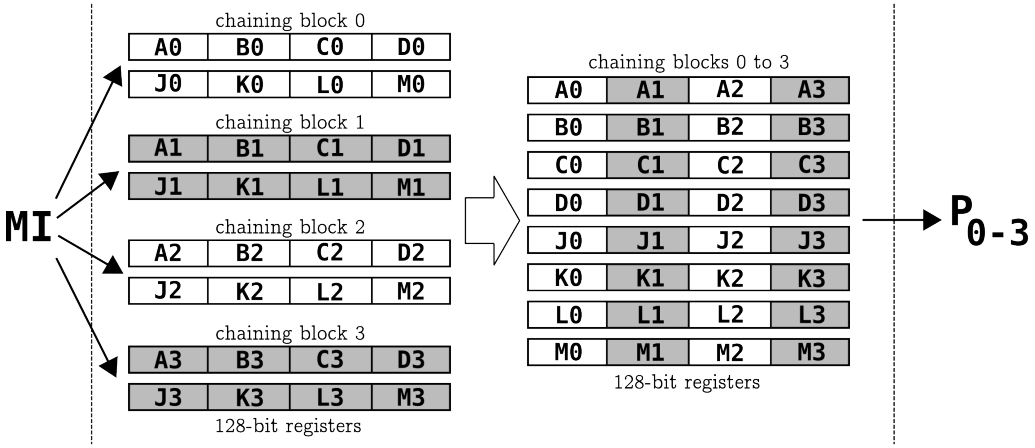

Figure 12. SSE parallelism

In MixWord function, we have four kinds of left rotations: 1, 2, 10 and 14 bits. Due to the SSE instructions one can rotate the four 32-bit words by n bits independently, without masks:

```
//r0 holds the register to be rotated
r0 = _mm_xor_si128(_mm_slli_epi32(r0, n), _mm_srli_epi32(r0, 32−n));
```

However, if we apply the perfect shuffle in the 128-bit register, interleaving the four 32-bit words, as shown in Figure 13, the MixWord left rotations will become 4, 8, 40 and 56. Three kinds of left rotations can be implemented through byte shuffle and a proper mask, at the cost of just one instruction. For example, an 8-bit (1 byte) left rotation:

```
//r0 holds the register to be rotated
mask = _mm_set_epi8(0x0E, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08, 0x07,
                    0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0F);

r0 = _mm_shuffle_epi8(r0, mask);
```

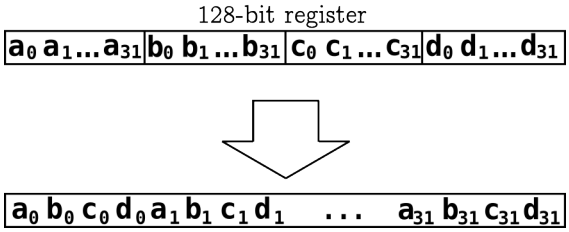

Figure 13. A perfect shuffle of four 32-bit words inside a 128-bit register

In this scenario we have made two kinds of implementations: applying perfect shuffle with the authors' parallel method and just applying perfect shuffle. In the former, the implementation is straightforward, we proceed such as the 64-bit scenario. However, in the latter, we must change the way of implementing the SubCrumb function. Due to the new bit arrangement inside the registers, we need to use PTLU through shuffle instructions instead of the logical operations suggested by the authors.

In addition, we have used a different approach to implement the multiplication by x computed in the message injection.

5.2.1. SubCrumb

SubCrumb receives eight 32-bit words as input, but the last four words are input in different order from the first four words.

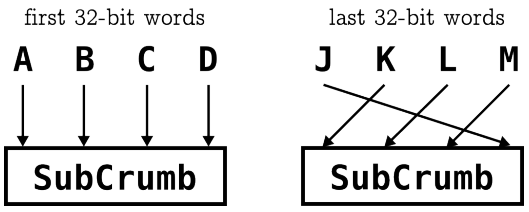

Figure 14. SubCrumb input order

Changing the order inside the register $[JKLM]$ was performance-infeasible, so we use two S-Box tables to deal with the different input orders. The SSE implementation in C code is shown in Appendix A.

5.2.2. Message Injection

Multiplication by x (0x02) in the ring $GF((2^{32})^8)$ with the definition polynomial $\phi(x) = x^8 + x^4 + x^3 + x + 1$ is frequently used in the message injection function (See Section 2).

An element $a = A + Bx + Cx^2 + Dx^3 + Jx^4 + Kx^5 + Lx^6 + Mx^7 \in GF((2^{32})^8)$ can be represented using two 128-bit SSE registers. The operation $a.x \mod \phi(x)$ can be computed as follows

$$\begin{aligned} a.x &= Ax + Bx^2 + Cx^3 + Dx^4 + Jx^5 + Kx^6 + Lx^7 + Mx^8 \\ &\quad \text{since } x^8 \equiv x^4 + x^3 + x + 1 \mod \phi(x) \text{ we have} \\ a.x &\equiv Ax + Bx^2 + Cx^3 + Dx^4 + Jx^5 + Kx^6 + Lx^7 \\ &\quad + M(x^4 + x^3 + x + 1) \mod \phi(x) \\ &\equiv M + (A + M)x + Bx^2 + (C + M)x^3 + (D + M)x^4 \\ &\quad + Jx^5 + Kx^6 + Lx^7 \mod \phi(x). \end{aligned}$$

In Figure 15 we depicted the multiplication by x. In terms of SSE registers, the implementation involves some shifting and XOR of the 32-bit words.

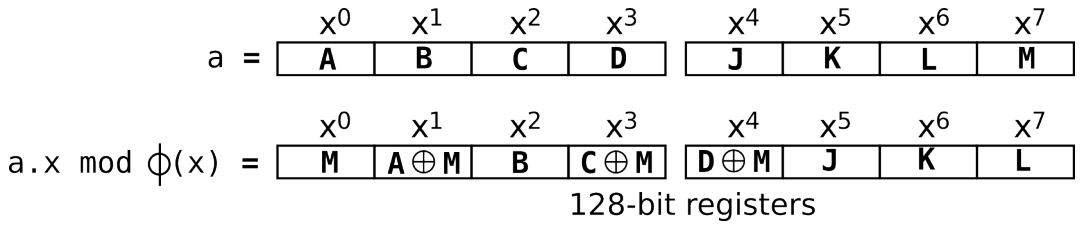

Figure 15. Multiplication $a.x \mod \phi(x)$ represented by 128-bit registers

When an element a is represented using the perfect shuffle operation, the implementation of $a.x$ is a little more complex, requiring some extra bit-level operations. However, the total cost increase is not substantial. The C code of our implementation using SSE intrinsics, is found in Appendix B.

We achieved good performance with Luffa-256, obtaining 15% of speed improvement with the Intel Compiler (ICC). The Luffa-512 could be implemented by mixing the two implementations; the first four blocks were implemented with perfect shuffle and parallelism, while the last block was better suited to be executed with perfect shuffle only.

6. Experimental Results

This section presents the results of our Luffa implementations for 64-bit and SSE platforms. The code was written in C, using intrinsics for SSE instructions. The programs were compiled with ICC v.11.1, GCC v.4.3 (Ubuntu 10.04 64-bit, kernel 2.6.32-24) and Visual Studio 2010 C++ Compiler (Windows Vista Business Edition 64-bit); they were executed in a Intel Core 2 Duo T6400 2.00GHz machine. Compiler flags included optimization level `-O2` for ICC, `-O3` for GCC and tuning with `-march=core2` switch. For 64-bit platform the flag `-no-vec` (`-mno-sse` for GCC) was used to prevent the compilers from using SSE vector instructions. The Visual Studio C++ compiler were executed with the flags `\O2`, `\GL` and tuned with `\favor:INTEL64`.

We noted little differences between the compilers: GCC requires more code tunning and produces less efficient programs on SSE platform. On 64-bit platforms, the three compilers present similar behaviors. We have chosen GCC due to its popularity and also because it is frequently chosen for benchmarks. ICC was selected because of its good performance using SSE instructions. Visual Studio, the NIST recommendation, has different performance on the SSE platform. In particular, the Perfect Shuffle implementation was the fastest for every version of the Luffa family.

We got running times for the following four versions of Luffa for 64 bits: Reference, the code provided by the authors in their submission package for NIST ("ANSI C" in 64-bit mode) [Canniere et al. 2009b]; Basic, our implementation based on the optimization used in the authors' code; Parallel, our implementation that uses two 32-bit words in a 64-bit register to execute two permutations in parallel; and Parallel+PS, our implementation based on the perfect shuffle operation combined with the execution of two permutations in parallel.

As a matter of consistency, our new implementation performances are compared with our Basic implementation. In addition, we give the performance of the authors' code executed in our machine, due to the fact that it is the reference code used by NIST for its computational efficiency evaluations. In Tables 3 and 4, are given the preformance results for three families of Luffa: Luffa-256, Luffa-384 and Luffa-512. Luffa-224 has the same algorithm as Luffa-256, with a chopping before the message digest output. Bold text represents the fastest implementation.

Table 3. 64-bit platform (cycles/byte), very long message on Intel Core 2 Duo

Implementation	ICC			GCC			Visual Studio 2010		
	256	384	512	256	384	512	256	384	512
Reference [Canniere et al. 2009b]	26.81	41.46	57.31	27.65	41.80	58.49	27.46	42.03	57.90
This work:									
Basic	24.31	33.09	41.50	24.43	34.35	46.35	25.34	36.37	46.03
Parallel	33.25	35.28	52.25	35.58	36.83	56.60	51.53	53.81	79.75
Parallel+PS	**20.00**	**22.47**	**35.09**	**20.39**	**23.89**	**35.85**	**22.62**	**25.37**	**37.34**

It can be noticed the overhead occurred when we work in parallel. This overhead is considerably reduced when we apply perfect shuffle, obtaining performance improvements up to 32% (Luffa-384 ICC) over the Basic implementation.

The SSE implementations are presented in the following way: Reference, the code provided by the authors in their submission package for NIST ("C using SSE intrinsics" in 64-bit mode); Basic[1], our implementation based on the optimization used in the authors' code; PS, our implementation based on the perfect shuffle operation; and Parallel+PS, our implementation with the perfect shuffle combined with the execution of four permutations in parallel.

We could achieve up to 15% of performance improvement (Luffa-256 ICC), and a new way to represent the SubCrumb functions, exploring the SSE instructions. We

[1]The implementation of the MixWord function was modified to improve the optimizations realized by the Visual Studio Compiler. As a consequence, the codes generated by ICC and GCC became a little slower.

Table 4. SSE platform (cycles/byte), very long message on Intel Core 2 Duo

Implementation	ICC			GCC			Visual Studio 2010		
	256	384	512	256	384	512	256	384	512
Reference [Canniere et al. 2009b]	17.96	20.93	31.34	16.59	17.96	32.03	17.15	19.09	28.62
This work:									
Basic	13.84	**14.78**	19.87	**15.28**	**16.18**	**24.81**	20.06	21.28	27.18
PS	**11.75**	16.22	22.56	15.31	20.90	28.15	**15.03**	**18.40**	**23.84**
Parallel+PS	12.34	14.96	**19.81***	15.81	18.15	25.31*	20.81	24.18	30.09*

(*) This implementation uses a mix of PS and Parallel+PS. See Section 5.

consider that PS implementation allowed ICC and Visual Studio to do a better code optimization. GCC could not profit from this new method and generated codes with similar performance. It can be observed that for only the Visual Studio compiler, the Reference code outperformed our Basic implementation.

It is presented in Table 5 a comparison of the code size of Reference and PS implementations. It can be observed a length reduction on 64-bit platform, due to the rotation instructions, and a little increase on SSE platform.

Table 5. Binary sizes of 64-bit and SSE implementations compiled in ICC (Kb)

	64-bit		SSE	
	Parallel	Parallel+PS	Basic	PS
Luffa-256	39.175	36.167	35.136	37.057
Luffa-384	35.428	36.071	35.133	37.043
Luffa-512	39.239	39.282	35.598	40.599

7. Conclusion

This paper presented a fast software implementation of Luffa for the Intel Core Duo processor. We have shown a novel application of the perfect shuffle operation to reduce the computational cost of the bit left rotations used in the MixWord function. We have also described a new way to implement Luffa using the vector SSSE3 instruction set; in particular, our implementation of the SubCrumb function takes advantage of the perfect shuffle operation and the shuffle instruction. The performance numbers of our 64-bit implementations show an improvement of 16 to 32% compared to the Reference [Canniere et al. 2009b] implementation.

As further research, the application of the perfect shuffle operation for low-cost 8-bit and 16-bit CPUs. In addition, we expect to apply the optimization techniques of this work to other hash algorithms.

References

Bernstein, D. J. and Lange, T. eBACS: ECRYPT benchmarking of cryptographic systems. `http://bench.cr.yp.to`. Accessed 15 July 2010.

Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. (2010). Keccak sponge function family main document version 2.1. `http://keccak.noekeon.org/Keccak-main-2.1.pdf`.

Bertoni, G., Daemen, J., Peeters, M., Assche, G. V., Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. (2007). Sponge functions. Ecrypt Hash Workshop 2007. Also available as public comment to NIST from http://www.csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html.

Biham, E., Chen, R., Joux, A., Carribault, P., Lemuet, C., and Jalby, W. (2005). Collisions of SHA-0 and reduced SHA-1. In *EUROCRYPT '05*, volume 3494 of *LNCS*, pages 36–57. Springer-Verlag.

Canniere, C. D., Sato, H., and Watanabe, D. (2009a). Hash function Luffa: Specification 2.0.1. Submission to NIST (Round 2). http://www.sdl.hitachi.co.jp/crypto/luffa/Luffa_v2_Specification_20091002.pdf.

Canniere, C. D., Sato, H., and Watanabe, D. (2009b). Hash function Luffa: Supporting document. Submission to NIST (Round 2). http://www.sdl.hitachi.co.jp/crypto/luffa/Luffa_v2_SupportingDocument_20090915.pdf.

Intel (2002). Intel architecture software developer's manual volume 2: Instruction set reference. http://www.intel.com.

Intel (2010). Intel Advanced Vector Extensions programming reference. http://www.intel.com.

Knežević, M. and Verbauwhede, I. (2009). Hardware evaluation of the Luffa hash family. In *WESS '09: Proceedings of the 4th Workshop on Embedded Systems Security*, pages 1–6, New York, NY, USA. ACM.

Kobayashi, K., Ikegami, J., Matsuo, S., Sakiyama, K., and Ohta, K. (2010). Evaluation of hardware performance for the SHA-3 candidates using SASEBO-GII. Cryptology ePrint Archive, Report 2010/010. http://eprint.iacr.org/.

Namin, A. H. and Hasan, M. A. (2009). Hardware implementation of the compression function for selected SHA-3 candidates. Technical Report from CACR 2009-28. http://www.cacr.math.uwaterloo.ca/techreports/2009/cacr2009-28.pdf.

NIST (2002). Secure Hash Standard, Federal Information Processing Standards publication, FIPS pub 180-2. Technical report, Department of Commerce. http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.

Oikawa, K., Wang, J., Kodama, E., and Takata, T. (2010). Implementation and evaluation of cryptographic algorithm on OpenCL. SCIS 2010, 3C4-2 (in Japanese).

Pornin, T. Software library Sphlib 2.1. http://www.saphir2.com/sphlib/. Accessed 26 July 2010.

Pornin, T. (2010). Comparative performance review of most of the SHA-3 second-round candidates. http://tahoe-lafs.org/~zooko/report-speed-sha3.pdf.

Tillich, S., Feldhofer, M., Kirschbaum, M., Plos, T., Schmidt, J.-M., and Szekely, A. (2009). High-speed hardware implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein. Cryptology ePrint Archive, Report 2009/510. http://eprint.iacr.org/.

Wang, X., Yin, Y. L., and Yu, H. (2005). Finding collisions in the full SHA-1. In *CRYPTO '05*, volume 3621 of *LNCS*, pages 17–36. Springer-Verlag.

Warren, H. S. (2002). *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Appendix A SubCrumb implementation for SSE platform with perfect shuffle

```
tbl00 = _mm_set_epi8(0x04,0x02,0x08,0x0F,0x0C,0x09,0x03,0x0B,
                     0x06,0x07,0x0A,0x05,0x01,0x00,0x0E,0x0D);
tbl01 = _mm_set_epi8(0x40,0x20,0x80,0xF0,0xC0,0x90,0x30,0xB0,
                     0x60,0x70,0xA0,0x50,0x10,0x00,0xE0,0xD0);

tbl10 = _mm_set_epi8(0x08,0x0C,0x04,0x0E,0x01,0x05,0x0F,0x0A,
                     0x09,0x02,0x03,0x00,0x06,0x0D,0x07,0x0B);
tbl11 = _mm_set_epi8(0x80,0xC0,0x40,0xE0,0x10,0x50,0xF0,0xA0,
                     0x90,0x20,0x30,0x00,0x60,0xD0,0x70,0xB0);

mask = _mm_set_epi32(0x0F0F0F0F, 0x0F0F0F0F, 0x0F0F0F0F, 0x0F0F0F0F);

void SubCrumb_128(__m128i *r0, __m128i tbl0, __m128i tbl1) {
        __m128i tmp;

        tmp = _mm_and_si128(*r0, mask);
        tmp = _mm_shuffle_epi8(tbl0, tmp);
        *r0 = _mm_srli_epi32(*r0, 4);
        *r0 = _mm_and_si128(*r0, mask);
        *r0 = _mm_shuffle_epi8(tbl1, *r0);
        *r0 = _mm_xor_si128(*r0, tmp);
}
```

Appendix B Multiplication by x in $GF((2^{32})^8)$ for SSE platform with perfect shuffle

```
mask = _mm_set_epi32(0x88888888, 0x88888888, 0x88888888, 0x88888888);

static void Mult0x02(__m128i *r0, __m128i *r1) {
        __m128i tmp0, tmp1;

        tmp0 = _mm_and_si128(*r0, mask); //D bits
        tmp1 = _mm_and_si128(*r1, mask); //M bits
        *r0 = _mm_xor_si128(*r0, tmp0);
        *r1 = _mm_xor_si128(*r1, tmp1);
        *r0 = _mm_slli_epi32(*r0, 1);
        *r1 = _mm_slli_epi32(*r1, 1);

        tmp0 = _mm_srli_epi32(tmp0, 3);
        *r1 = _mm_xor_si128(*r1, tmp0);
        *r0 = _mm_xor_si128(*r0, tmp1);

        tmp1 = _mm_srli_epi32(tmp1, 2);
        *r0 = _mm_xor_si128(*r0, tmp1);

        tmp1 = _mm_srli_epi32(tmp1, 1);
        *r1 = _mm_xor_si128(*r1, tmp1);
        *r0 = _mm_xor_si128(*r0, tmp1);
}
```