Journal of Information Security, 2013, 4, 7-11
http://dx.doi.org/10.4236/jis.2013.41002 Published Online January 2013 (http://www.scirp.org/journal/jis)

A *j*-Lanes Tree Hashing Mode and *j*-Lanes SHA-256

Shay Gueron[1,2]
[1]Department of Mathematics, University of Haifa, Haifa, Israel
[2]Intel Corporation, Israel Development Center, Haifa, Israel
Email: shay@math.haifa.ac.il

Received August 19, 2012; revised September 20, 2012; accepted October 12, 2012

ABSTRACT

j-lanes hashing is a tree mode that splits an input message to *j* slices, computes *j* independent digests of each slice, and outputs the hash value of their concatenation. We demonstrate the performance advantage of *j*-lanes hashing on SIMD architectures, by coding a 4-lanes-SHA-256 implementation and measuring its performance on the latest 3rd Generation Intel® Core™. For messages whose lengths range from 2 KB to 132 KB, we show that the 4-lanes SHA-256 is between 1.5 to 1.97 times faster than the fastest publicly available implementation that we are aware of, and between ~2 to ~2.5 times faster than the OpenSSL 1.0.1c implementation. For long messages, there is no significant performance difference between different choices of *j*. We show that the 4-lanes SHA-256 is faster than the two SHA3 finalists (BLAKE and Keccak) that have a published tree mode implementation. Finally, we explain why *j*-lanes hashing will be faster on the coming AVX2 architecture that facilitates using 256 bits registers. These results suggest that standardizing a tree mode for hash functions (SHA-256 in particular) could be useful for performance hungry applications.

Keywords: Tree Mode Hashing; SHA-256; SHA3 Competition; SIMD Architecture; Advanced Vector Extensions Architectures; AVX; AVX2

1. Introduction

The performance of hash functions plays an important role in various situations (e.g., for SSL/TLS connections that use HMAC for authenticated encryption). In particular, the performance of SHA-256 on high end processors is a performance baseline for the SHA3 competition [1].

Recently, [2] published a "Simultaneous Hashing" (S-HASH) method, for using SIMD architectures to speed up the computations of SHA-256 (and other hashes) over multiple messages. In this paper, we apply this technique to accelerate SHA-256 for a single message, using a tree mode that we call *j*-lanes hashing. We show that the resulting "*j*-lanes SHA-256" is significantly faster than the standard ("linear" hereafter) SHA-256. This demonstrates the performance benefits that applications could gain if a tree mode for hash functions (in particular, SHA-256 and SHA-512) is standardized. It is interesting to compare our results to the two SHA3 finalists that already have a *j*-lanes tree mode ($j = 2$) implementation (see [3]): BLAKE and Keccak. We offer this comparison in Section 3.

2. *j*-Lanes Hashing and the Special Case of 4-Lanes SHA-256

Tree hashing is a well known concept for efficient computations of hash functions, and is an efficient way for updating a hash value when only a portion of the message is changed. Some relevant references are [4-8]. We focus here on a specific tree construction, which is defined in the following section.

2.1. *j*-Lanes Hashing

Definition 1 (message *j*-Slicing): given a message m, its associated j-Sliced message is obtained by applying a permutation to the bits of m, followed by slicing the resulting message into consecutive disjoint slices. We denote this by $m = permutation\ (m_1 \| m_2 \| m_3 \| \ldots \| m_j)$ under some agreed convention on how each slice is defined, and what the permutation is (for simplicity we assume that m has at least j bits, to avoid degenerate "null" slices).

Definition 2 (*j*-lanes-hash): Let $h = h\ (MESSAGE)$ be a hash function. Its associated j-lanes-hash, is a hash scheme that operates as follows:

1) j-Slicing the message to $m = permutation\ (m_1 \| m_2 \| m_3 \| \ldots \| m_j)$.

2) Computing $t_1 = h\ (m_1)$, $t_2 = h\ (m_2)$, …, $t_j = h\ (m_j)$.

3) Computing $t^* = h\ (t_1 \| t_2 \| \ldots \| t_j)$.

4) Returning the digest t^*.

(hereafter we call Step 3 the "Wrapping" step).

j-lanes-hash is a special form of a tree mode (not a binary tree), where the number of nodes is $j + 1$ and the height of the tree is 2. As a special case of a tree mode, the security properties of this construction follow from the more general theory on tree hashing (e.g., [5,6] discuss the security properties of a tree hash in the context of indifferentiability from an ideal hash function).

Note that the above definition is flexible enough to cover several useful setups. One example is "interleaving" segments of a given message (which we use here, for directly taking advantage of SIMD architectures). Another case is when the data is consumed from j locations (e.g., j pointers) of a message. This can occur, for example, in an application that hashes a file system (or a directory) where j is the number of files (and each file is a node in the tree).

Hereafter, we assume that the processed messages are sufficiently long to gain performance from the j-lanes tree mode (and ignore trivially short message).

2.2. Applying *j*-Lanes Hashing to Derive a 4-Lanes SHA-256

We use SHA-256 as the underlying hash algorithm, and generate a "*j*-lanes SHA-256". Our motivation is to check the potential performance advantage of the parallelization supported by SIMD architectures (or multithreaded implementations).

By splitting the message into j independent slices, the hash computations are reduced to the problem of hashing multiple independent messages, supplemented by the fixed-cost Wrapping step. With this, we can apply the techniques for utilizing SIMD architectures to hash multiple independent messages (of different lengths). These techniques, and their resulting performance, are described in detail in [2].

SHA-256 operates on 32-bit words. Therefore, on processors that support the AVX (or SSE) architecture that has 128-bit registers and the necessary integer instructions, a natural choice for j-lanes (SHA-256) hashing is $j = 4$, with the obvious convention for slicing the message: consecutive 128-bit chunks of the message are treated as 4 consecutive 32-bit words, each one belongs to a different slice. These 4 words can then be viewed as 4 "elements" of a single AVX register (*xmm*), and the SHA-256 computations can be parallelized using the SIMD architecture (see [2] for details). If the byte-length of the message is divisible by 256, we set the slices to have equal lengths. Otherwise, (at least) one of slice has a different length. This situation requires different handling in the last Update (with negligible performance cost).

j-lanes hashing involves some overhead, and therefore, the performance gains are expected to be (fully) manifested only for sufficiently long messages.

To illustrate, we note that the performance of SHA-256 is closely proportional to the number of invocations of its compression function ("Update" hereafter). Consider a message whose byte-length l is divisible by 256, and write $l = 256x$ for some integer x. Hashing (with SHA-256) such a message requires $4x + 1$ Updates, where the last one due to the padding block. On the other hand, 4-lanes SHA-256 for this message requires $4(x + 1) + 3$ Updates, accounting for 1 padding block for each slice, and 3 Updates for the Wrapping step which requires hashing of a 128 bytes message. Comparing the linear (*i.e.*, serial) SHA-256 to the 4-lanes SHA-256, we see that the latter involves 6 additional Updates. However, from the total of $4x + 7$ Updates, $4x$ can be parallelized, in particular by using the AVX architecture. This is the reason why the overall performance is expected to improve.

3. Performance Studies

This section discusses some performance studies for of j-lanes SHA-256. We first describe the measurement methodology.

- Each measured function was isolated, run 25,000 times (warm-up), followed by 100,000 iterations that were timed (using the RDTSC instruction) and averaged.
- To minimize the effect of background tasks running on the system, each experiment was repeated five times, and the minimum result was recorded.
- All the runs were carried out on a system where the Intel® Hyper-Threading Technology, the Intel® Turbo Boost Technology, and the Enhanced Intel Speedstep® Technology, were *disabled*.
- The runs were executed on the 3rd Generation Intel® Core™ i7-3770 processor (previously known as "Architecture Code name Ivy Bridge").
- In all cases, the reported performance numbers account for the full computations (*i.e.*, including the padding and, when relevant. the final hashing of the j digests).
- In our studies, we used two SHA-256 and three j-lanes-SHA-256 ($j = 4, 8, 16$) implementations as follows:
- OpenSSL (1.0.1c) linear: standard hashing using OpenSSL function.
- 4-SMS linear: standard hashing using the n-SMS ($n = 4$) method (see [9,10]; we used here an improved version of this implementation).
- j-lanes using OpenSSL: using OpenSSL's (1.0.1c) SHA-256 function to implement j-lanes-SHA-256.
- j-lanes using the n-SMS: using the n-SMS SHA-256 implementation ([9,10]) to implement j-lanes SHA-256.

- AVX *j*-lanes hashing (*j*-lanes hashing for short): an optimized implementation of *j*-lanes SHA 256, using the S-HASH implementation of [2], and the AVX architecture.

The results are illustrated in **Figures 1-3**.

Figure 1 compares the different implementations for an 8 KB message and *j* = 4. Without parallelizing the hashing of the slices (as in *j*-lanes using OpenSSL and *j*-lanes using the *n*-SMS), the 4-lanes SHA-256 is slower than the linear implementation. This is due to the overheads of the *j*-lanes method. For example, OpenSSL (1.0.1c) uses 129 Updates and performs at 12.87 Cycles/Byte, while the 4-lanes SHA-256 implementation that simply calls the OpenSSL functions, uses 135 Updates, and performs at 13.57 Cycles/Bye. On the other hand, the optimized (using AVX) 4-lanes SHA-256 implementation is 2.45 times faster than OpenSSL.

Figure 2 illustrates the effect of the choice of *j* (= 4, 8, 16). Obviously, increasing *j* involves additional overhead to the *j*-lanes hashing. For example, 16-lanes SHA-256 for an 8 KB message involves 153 Updates, and is therefore slower than the 4-lanes SHA-256 that uses only 135 Updates (see top panel). However, both 8-lanes and 16-lanes SHA-256 are still significantly faster than the best performing linear implementation. For long messages (see bottom panel), the relative impact of the overheads decreases, and we obtain roughly the same performance for *j* = 4, 8, 16.

Figure 3 shows the performance advantage of the 4-lanes SHA-256 for messages of lengths varying from 2 KB to 128 KB: 4-lanes SHA-256 is between 1.55 to 2 times faster than the best serial implementation (and 1.97 - 2.53 times faster than OpenSSL 1.0.1c).

Figure 4 shows the performance advantage of the 4-lanes SHA-256 for a long message of 1 MB, measured on

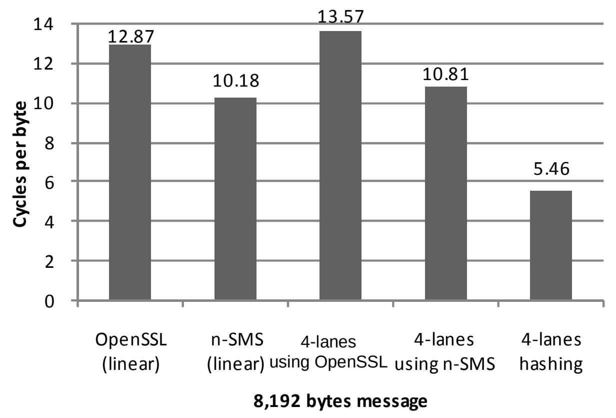
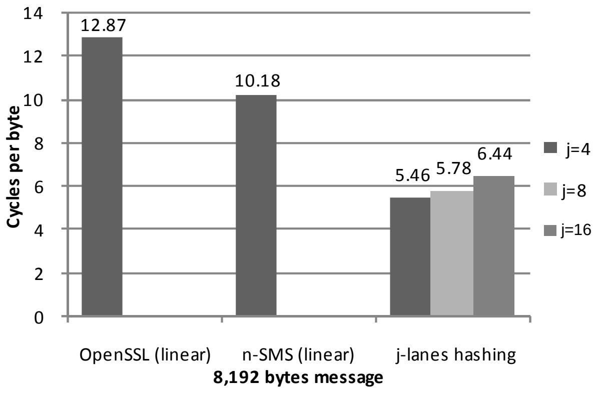
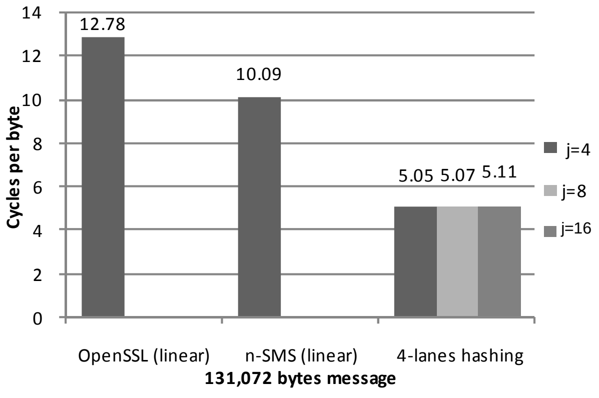


Figure 1. Performance of different implementations of 4-lanes SHA-256, compared to linear SHA-256, for a 8192-byte message. Measurements taken on the 3rd Generation Intel® Core™ Processor.



(a)



(b)

Figure 2. Performance of *j*-lanes-SHA-256 for *j* = 4, 8, 16, compared to linear SHA-256. The message length is 8192 bytes (top panel) and 131,072 bytes (bottom panel). Measurements taken on the 3rd Generation Intel® Core™ Processor.

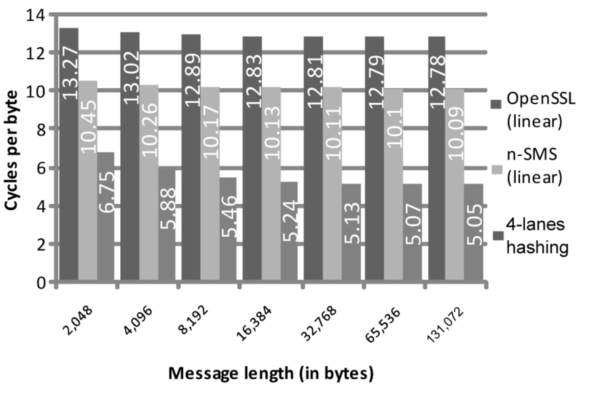


Figure 3. Performance of 4-lanes SHA-256, compared to linear SHA-256, for different message lengths. Measurements taken on the 3rd Generation Intel® Core™ Processor.

the two processors: the 2nd Generation and the 3rd Generation Intel® Core™ Processors. We can see that the 4-lanes SHA-256 is 2.27*x* faster than the best linear implementation of SHA-256, when measured on the 2nd Generation Intel processor, and 2*x* faster when measured

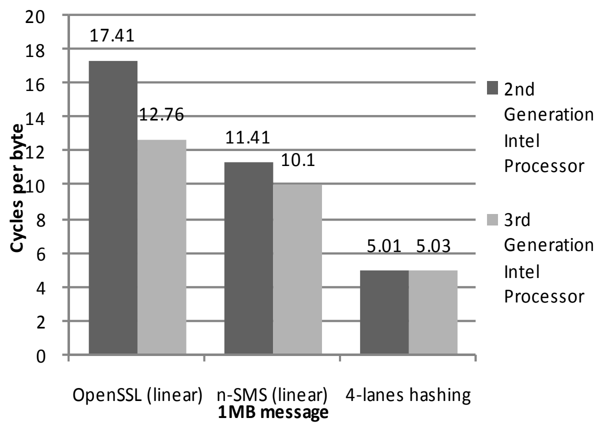


Figure 4. Performance of 4-lanes SHA-256, compared to linear SHA-256, for a 1 MB message. Measurements taken on the 2nd Generation and on the 3rd Generation Intel® Core™ Processor.

on the 3rd Generation Intel processor.

A Comment on the SHA3 Finalists

We expect that the SHA3 finalists [1] could also gain from using the *j*-lanes-hash, at least to some extent, and the performance gains will further increase when the AVX2 architecture becomes available. However, at this point, it is hard to tell if these algorithms would outperform the *j*-lanes-SHA-256 and/or *j*-lanes-SHA-512, and by what margin.

Since the two finalists BLAKE and Keccak already have a tree mode implementation (j = 2 for BLAKE and Keccak; see [3]), we show the performance comparisons of SHA-256, BLAKE, and Keccak in linear and in *j*-lanes mode in **Figure 5**. Although we tried a 4-lanes BLAKE512, we could not get it to perform better than the underlying hash function. This is due to the fact that the optimized BLAKE512 implementation already exploits the parallelism offered by the AVX architecture.

As expected, the *j*-lanes (tree mode) implementation improves the performance of all three algorithms. The results show that the *j*-lanes SHA-256 implementation is the fastest one of these three.

Recalling that SHA-256 (and SHA-512) is the performance baseline for SHA3, we conclude (from the currently available information) that considering the *j*-lanes mode still *does not* offer a performance advantage for SHA3 over SHA-256. This is consistent with the findings of [6]: migration to a new SHA3 standard could not be motivated by performance advantages on the high end platforms[1].

[1]Since the paper was submitted, NIST has announced Keccak as the winner of the SHA3 competition. The performance analysis (and comparison) that is provided here is still useful for assessing the implications of this selection (from the performance viewpoint).

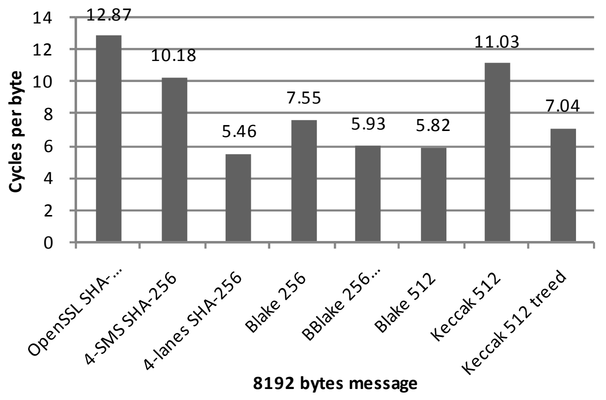


Figure 5. Performance of SHA-256, BLAKE256, and Keccak, in "linear" mode and in tree mode (for a 8192-byte message). BLAKE512 is brought here only for comparison (4-lanes BLAKE512 is not shown because we could not gain good performance from this implementation). Measurements taken on the 3rd Generation Intel® Core™ Processor.

4. Conclusions

We demonstrated the performance gains of *j*-lanes hashing, using SHA-256 as the underlying hash algorithm. On the 3rd Generation Intel® Core™ Processor (with AVX architecture) selecting j = 4, gives speedup factors between 1.55*x* to 2*x*, compared to the best available implementation (up to 2.53*x* when comparing to OpenSSL 1.0.1c). We focused on j = 4, as the natural choice for the current AVX (and SSE) architectures. Interestingly, although j = 4 yields the best results (the Wrapping overhead is the smallest among the tested cases), we note that the performance with all the studies choices j = 4, 8, 16 is roughly the same for long messages.

We also comment that with the near future AVX2 architecture [11], a natural choice would be j = 8 for SHA-1, SHA-256 and j = 4 for SHA-512, and the *j*-lanes implementations will be significantly faster. Therefore, if a *j*-lanes hashing mode is adopted, and the ecosystem would prefer to support only a single value of *j* (to reduce the interoperability complexities), it seems that selecting j = 8 would be a good choice.

In general, the *j*-lanes-hash can be useful in other scenarios, and with different values of *j*. One example mentioned about is hashing a file system, where *j* is the number of files (and each file is a node in the tree). Such computations can be accelerated not only by using SIMD architectures, but also by using the processing power of multi-cores systems.

We conclude that the *j*-lanes-hash could alleviate computational bottlenecks, and recommend that this mode (or a general tree mode) should be standardized. To this end, we comment that standardization of a *j*-lanes (or any tree) mode should also properly define different initialization vectors (depending also on the value of *j*) in order to distinguish the resulting digests from outputs of the linear SHA-256 (analogously to the how a digest truncation (e.g., SHA-224) is defined).

5. Acknowledgements

I thank Jean-Philippe Aumasson, Bart Preneel, and Jesse Walker for helpful discussions.

REFERENCES

[1] National Institute of Standards and Technology, "Cryptographic Hash Algorithm Competition," 2005. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

[2] S. Gueron and V. Krasnov, "Simultaneous Hashing of Multiple Messages", *Journal of Information Security*, Vol. 3, 2012, pp. 319-325. doi:10.4236/jis.2012.34039

[3] Virtual Applications and Implementations Research Lab, "SUPERCOP," 2012. http://bench.cr.yp.to/supercop.html

[4] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, "Keccak Sponge Function Family Main Document," 2009. http://cuda-keccak.googlecode.com/svn/trunk/docs/Keccak-main-2.1.pdf

[5] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, "Sufficient Conditions for Sound Tree and Sequential Hashing Modes," 2009. http://eprint.iacr.org/2009/210

[6] Y. Dodis, L. Reyzin, R. L. Rivest and E. Shen, "Indifferentiability of Permutation-Based Compression Functions and Tree-Based Modes of Operation, with Applications to MD6," *Proceedings of FSE* 2009, *Lecture Notes in Computer Science*, Leuven, 22-25 February 2009, pp. 104-121.

[7] R. C. Merkle, "A Certified Digital Signature," *Advances in Cryptology—Proceedings of CRYPTO*'89, *Lecture Notes in Computer Science*, Santa Barbara, 20-24 August 1989, pp. 218-238.

[8] P. Sarkar and P. J. Schellenberg, "A Parallelizable Design Principle for Cryptographic Hash Functions," 2002. http://eprint.iacr.org/2002/031

[9] S. Gueron and V. Krasnov, "Parallelizing Message Schedules to Accelerate the Computations of Hash Functions," *Journal of Cryptographic Engineering*, Vol. 4, No. 4, 2012, pp. 241-253.

[10] S. Gueron and V. Krasnov, "[PATCH] Efficient Implementations of SHA256 and SHA512, Using the Simultaneous Message Scheduling method," 2012. http://rt.openssl.org/Ticket/Display.html?id=2784&user=guest&pass=guest

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