Vectorization of ChaCha Stream Cipher

Martin Goll[1], Shay Gueron[2,3]

[1] Ruhr-University Bochum, Germany
[2] Department of Mathematics, University of Haifa, Israel
[3] Intel Corporation, Israel Development Center, Haifa, Israel

Abstract. This paper describes software optimization for the stream Cipher ChaCha. We leverage the wide vectorization capabilities of the new AVX2 architecture, to speed up ChaCha encryption (and decryption) on the latest x86_64 processors. In addition, we show how to apply vectorization for the future AVX512 architecture, and get further speedup. This leads to significant performance gains. For example, on the latest Intel Haswell microarchitecture, our AVX2 implementation performs at 1.43 cycles per byte (on a 4KB message), which is ~2x faster than the current implementation in the Chromium project.

Keywords: Stream Cipher, ChaCha, SSL, TLS, optimization, Haswell

1 Introduction

Secure communication on the internet requires that the communication endpoints use different cryptographic primitives to establish a protected channel, and a common protocol to apply these primitives. The leading protocol for secure communication specifications is TLS [1]. TLS supports a diversity of public key algorithms for establishing a symmetric session key for two communication endpoints, and a variety of symmetric ciphers and MAC algorithms for the subsequent encrypted and authenticated communication. The performance of these primitives is crucial for efficient communication.

Currently [2], the most popular ciphers of the TLS protocol are RC4 and AES-CBC (with some hash based authentication), but they have been scathed by some problems/attacks ([3], [4]) .The AES-CBC issues have been fixed in TLS 1.1, but the fix is complex. The RC4 cipher is perceived unsecure. The problems with the existing ciphers result in strong motivation for developing and using new cipher suites.

A very promising alternative is the AES-GCM [5] authenticated cipher, whose software implementation has been extensively optimized [6]. This, however, implies that two secure cipher alternatives (in TLS) are based on the same cryptographic primitive AES, and past experience shows that this can be problematic. Consequently, it is useful to have a choice between different cryptographic primitives for the same purpose, and this is where ChaCha [7] and Poly1305 [8] become important. They are secure, relatively fast, and already have high quality public domain implementations. They also are naturally “constant time”, and have nearly perfect key agility. These properties led to the newly proposed TLS draft [9] which includes ChaCha20 as a stream cipher, and Poly1305 as the authenticator.

ChaCha has naturally good performance, and already has implementations that use 128-bit vectorization. In this paper, we show how to achieve higher performance by using wider vectorization: 256-bit AVX2 [10] instructions that are available on the new Haswell architecture, and 512-bit AVX512 [11] on future architectures.

2 Preliminaries

ChaCha is a 256-bit stream cipher, based on the Salsa20 [12] stream cipher. Compared to Salsa20, ChaCha has better diffusion per round and conjecturally increasing resistance to cryptanalysis. The core of the Salsa20 (and ChaCha) function is a hash function which maps 64 input bytes to a unique and irreversible 64-byte output keystream. Its 64-bit block counter restricts the maximum number of blocks for the output keystream to 2^{64} (i.e., a maximum keystream of 2^{40} GB). The encryption and decryption is done by xor'ing the keystream with the input data. Two useful features of ChaCha are the possibility of output block generation at random positions, and the naturally constant time for processing stream blocks.

2.1 ChaCha's Matrix

The input to the ChaCha function is a 256-bit key, a 64-bit nonce and a 64-bit block counter. They are all treated as 32-bit integer arrays in little endian format. The input values and four 32-bit constants are arranged in a 4 x 4 matrix. The following matrix (Fig. 1) shows the initial state before the round function operates on it.

$$\begin{pmatrix} 0x61707865 & 0x3320646E & 0x79622D32 & 0x6B206574 \\ key[0] & key[1] & key[2] & key[3] \\ key[4] & key[5] & key[6] & key[7] \\ counter[0] & counter[1] & nonce[0] & nonce[1] \end{pmatrix}$$

Fig. 1. Initial state matrix of ChaCha.

2.2 ChaCha's Round Function

ChaCha's algorithm is defined for 20 rounds (maximum security), 8 rounds (maximum speed) or 12 rounds (balance between speed and security). The round function is split into two alternating functions: the row-round function for odd rounds (1, 3, …, 19) and the column-round function for even rounds (2, 4, …, 20). The algorithms are shown in Fig. 2.

```
Algorithm 1: ROWROUND
                 for odd rounds
Input:  x0,x1,x2,x3,(state matrix)
        x4,x5,x6,x7,
        x8,x9,xA,xB,
        xC,xD,xE,xF
Output: x0,x1,x2,x3,(updated
        x4,x5,x6,x7, state matrix)
        x8,x9,xA,xB,
        xC,xD,xE,xF
Flow
    QUARTERROUND(x0, x4, x8, xC);
    QUARTERROUND(x1, x5, x9, xD);
    QUARTERROUND(x2, x6, xA, xE);
    QUARTERROUND(x3, x7, xB, xF);
Return
```

```
Algorithm 2: COLUMNROUND
                 for even rounds
Input:  x0,x1,x2,x3,(state matrix)
        x4,x5,x6,x7,
        x8,x9,xA,xB,
        xC,xD,xE,xF
Output: x0,x1,x2,x3,(updated
        x4,x5,x6,x7, state matrix)
        x8,x9,xA,xB,
        xC,xD,xE,xF
Flow
    QUARTERROUND(x0, x5, xA, xF);
    QUARTERROUND(x1, x6, xB, xC);
    QUARTERROUND(x2, x7, x8, xD);
    QUARTERROUND(x3, x4, x9, xE);
Return
```

Fig. 2. Left panel: Row-round algorithm. Right panel: Column-round algorithm. Both algorithms apply the quarter-round function on the permuted state matrix.

Quarter-Round Function

The quarter-round function (Fig. 3) updates, reversibly, one row of the state matrix. The operations are 4 adds, 4 xors and 4 rotations, which are applied on the four 32-bit values of the row.

```
Algorithm 3: QUARTERROUND
Input:  a, b, c, d (32-bit row elements)
Output: a, b, c, d (updated 32-bit row elements)
Flow
        a += b; d ^= a; d <<<= 16;
        c += d; b ^= c; b <<<= 12;
        a += b; d ^= a; d <<<=  8;
        c += d; b ^= c; b <<<=  7;
Return
```

Fig. 3. Quarter-round algorithm.

Row-Round Function

The row-round function right-rotates the rows in the first step, and up-rotates the columns in the second step. The rotation count equals to the position of the row or the column. After both rotations, the row vectors are fed into the quarter-round function, as illustrated in Fig. 4.

$$\begin{pmatrix} x0 & x1 & x2 & x3 \\ x4 & x5 & x6 & x7 \\ x8 & x9 & xA & xB \\ xC & xD & xE & xF \end{pmatrix} \xRightarrow{row\ rot.} \begin{pmatrix} x0 & x1 & x2 & x3 \\ x7 & x4 & x5 & x6 \\ xA & xB & x8 & x9 \\ xD & xE & xF & xC \end{pmatrix} \xRightarrow{col.\ rot.} \begin{pmatrix} x0 & x4 & x8 & xC \\ x1 & x5 & x9 & xD \\ x2 & x6 & xA & xE \\ x3 & x7 & xB & xF \end{pmatrix}$$

Fig. 4. Row-round permutation of the state matrix.

On 32 (or less) bit microarchitectures, this row-round permutation is achieved at almost zero performance costs by using pointer arithmetic. On vector based architectures, the permutation requires real rotations.

Column-Round Function

The column-round function rotates the columns in the state matrix to the top by their position count. After the rotation the row vectors are fed into the quarter-round function (see Fig. 5).

$$\begin{pmatrix} x0 & x4 & x8 & xC \\ x1 & x5 & x9 & xD \\ x2 & x6 & xA & xE \\ x3 & x7 & xB & xF \end{pmatrix} \xRightarrow{\textit{column rotation}} \begin{pmatrix} x0 & x5 & xA & xF \\ x1 & x6 & xB & xC \\ x2 & x7 & x8 & xD \\ x3 & x4 & x9 & xE \end{pmatrix}$$

Fig. 5. Column-round permutation of the state matrix.

On 32 (or less) bit microarchitectures this column-round permutation is, similarly, nearly free. On vector based architectures, the vector elements must be reorganized between the row-round and column-round function. This requires real rotations.

2.3 ChaCha's Double-Round Function

The consolidation of the row-round function and the column-round function is called double-round function (see Fig. 6).

```
Algorithm4: DOUBLEROUND
Input:  x[16] (state matrix as array of 32-bit values)
Output: x[16] (updated state matrix)
Flow
        QUARTERROUND(x0, x4, x8, xC);
        QUARTERROUND(x1, x5, x9, xD);
        QUARTERROUND(x2, x6, xA, xE);
        QUARTERROUND(x3, x7, xB, xF);
        QUARTERROUND(x0, x5, xA, xF);
        QUARTERROUND(x1, x6, xB, xC);
        QUARTERROUND(x2, x7, x8, xD);
        QUARTERROUND(x3, x4, x9, xE);
Return
```

Fig. 6. Double-round algorithm, combines the row-round and the column-round functions.

2.4 Existing ChaCha Implementations

A variety of public domain ChaCha implementations are available. Some of them could be found on the author's webpage [13], and others appear in the eBACS project web page [14]. The NSS [15] and OpenSSL [16] libraries in the Chromium [17] project also include two implementations. We briefly summarize the main optimizations in these implementations.

128-Bit Vectorization

The row-round and the column-round functions execute, four times, the same quarter-round function, with four independent inputs. The input to a quarter-round is a four 32-bit element vector. This is used to calculate the four quarter-rounds in parallel with four 128-bit vectors as input (instead of four 32-bit values). This is illustrated in Fig. 7.

```
Algorithm 5: DOUBLEQUARTERROUND (optimized for 128-bit vectors)
Input:  v0, v1, v2, v3 (state matrix as four 4x32-bit vectors,
                        each vector includes one row)
Output: v0, v1, v2, v3 (updated state matrix)
Flow
       v0 += v1; v3 ^= v0; v3 <<<= (16, 16, 16, 16);
       v2 += v3; v1 ^= v2; v1 <<<= (12, 12, 12, 12);
       v0 += v1; v3 ^= v0; v3 <<<= ( 8,  8,  8,  8);
       v2 += v3; v1 ^= v2; v1 <<<= ( 7,  7,  7,  7);
       v1 >>>= 32; v2 >>>= 64; v3 >>>= 96;
       v0 += v1; v3 ^= v0; v3 <<<= (16, 16, 16, 16);
       v2 += v3; v1 ^= v2; v1 <<<= (12, 12, 12, 12);
       v0 += v1; v3 ^= v0; v3 <<<= ( 8,  8,  8,  8);
       v2 += v3; v1 ^= v2; v1 <<<= ( 7,  7,  7,  7);
       v1 <<<= 32; v2 <<<= 64; v3 <<<= 96;
Return
```

Fig. 7. 128-bit vector optimized double-quarter-round algorithm.

With the aggregated calculation of the four quarter-round functions, only one double-quarter-round per two rounds is left, and the load/store operations for the single 32-bit values in each quarter-round functions are replaced by one quarter load and store operations for 128-bit vectors. This comes at the cost of additional rotations that are needed for shuffling the order of the single elements in the vectors, to be suitable for either the row-round calculation or the column-round calculation.

3 Wider Vectorization

The purpose of this section is to show how to leverage the new AVX2 and the future AVX512 extensions to improve ChaCha's encryption/decryption performance.

3.1 256-Bit Vectorization

Compared to a 128-bit register, a 256-bit register can hold twice as many 32-bit values. This can be leveraged to store two row vectors at the same time, and operate on both of them, simultaneously. Therefore, the operational costs can be reduced from two load, two store and 2*X process instructions for two 128-bit vectors, to one load, one store and X process instructions for a 256-bit vector.

To take advantage of the 256-bit vector registers, we chose to process two ChaCha stream blocks simultaneously. We do not further cut down the count of row vector operations in the double-quarter-round because the remaining operations on the row vectors are mutually dependent. This makes it difficult to further reduce the number of vector operations through simultaneous calculations.

The updated double-quarter-round algorithm with the 256-bit vectorization is listed in Fig. 8.

```
Algorithm 6: DOUBLEQUARTERROUND (optimized for 256-bit vectors)
Input:  v0, v1, v2, v3 (2 state matrices as 4 8x32-bit vectors,
                        each vector includes one row
                        of each matrix)
Output: v0, v1, v2, v3 (updated state matrices)
Flow
       v0 += v1; v3 ^= v0; v3 <<<= (16,16,16,16,16,16,16,16);
       v2 += v3; v1 ^= v2; v1 <<<= (12,12,12,12,12,12,12,12);
       v0 += v1; v3 ^= v0; v3 <<<= ( 8, 8, 8, 8, 8, 8, 8, 8);
       v2 += v3; v1 ^= v2; v1 <<<= ( 7, 7, 7, 7, 7, 7, 7, 7);
       v1 >>>= 32; v2 >>>= 64; v3 >>>= 96;
       v0 += v1; v3 ^= v0; v3 <<<= (16,16,16,16,16,16,16,16);
       v2 += v3; v1 ^= v2; v1 <<<= (12,12,12,12,12,12,12,12);
       v0 += v1; v3 ^= v0; v3 <<<= ( 8, 8, 8, 8, 8, 8, 8, 8);
       v2 += v3; v1 ^= v2; v1 <<<= ( 7, 7, 7, 7, 7, 7, 7, 7);
       v1 <<<= 32; v2 <<<= 64; v3 <<<= 96;
Return
```

Fig. 8. 256-bit vector optimized double-quarter-round algorithm.

The improved double-quarter-round algorithm requires a change in the initialization of the vectors. We still use the 128-bit unaligned load instructions to transfer the input data, key and constant to the 256-bit vector registers. Then, we broadcast the 128-bit vectors to both 128-bit halves of the 256-bit vector (the “broadcast” instruction duplicates an element in a vector register). This requires four broadcast instructions, but it is still faster than loading the unaligned data in 256-bit blocks. The second state matrix also needs an incremented block counter. This can be computed by one extra vector addition with a constant (0,0,0,1).

Incrementing the block counter also needs to be adapted when more than two blocks are processed. This is done by changing the vector constant for 64-bit integers addition from (0,1) at 128-bit vectors to (0,2,0,2) at 256-bit vectors. This does not involve extra vector additions with constants.

Finally, xor’ing and writing of the encrypted (or decrypted) stream to the target buffer needs to be adjusted due to the order of the 128-bit rows in the four 256-bit vectors. Every one of the four 256-bit vectors includes, in the lower 128-bit part, the row vectors of the first output block (bytes 1-64), and includes, in the higher 128-bit part, the second output block (bytes 65-128). Therefore, we need four extra vector permutations to rearrange the bytes in the 256-bit vectors to the right stream order. Subsequently, we can save half of the xor, load and store operations, because we can operate on 256-bit instead of 128-bit blocks.

3.2 512-Bit Vectorization

The 512-bit vectorization extends the 256-bit vectorization in a way that four state matrices can be processed in parallel. Therefore, the double-quarter-round has to be adapted for 512-bit vectors. Similarly, the vector initialization changes from adding (0,1,0,0) to the initial 256-bit vector to adding (0,3,0,2,0,1,0,0) to the initial 512-bit vector. In addition, incrementing of the block counter changes from adding (0,2,0,2) to adding (0,4,0,4,0,4,0,4). Finally, the order of the row vectors in the 512-bit vectors needs to be adjusted again. As a result, we need 8 extra vector permutation instructions and 12 extra move instructions compared to the 256-bit implementation. However, working with 512-bit blocks, saves half the number of xor, load and store operations, and the overall performance is expected to be enhanced.

4 Results

This section shows the performance of our proposed optimizations. We compare our implementation to the vectorized ChaCha implementation of the NSS and OpenSSL sources from the latest development branch of the Chromium project (November 10, 2013; retrieved from the 'svn' repository of Chromium [18]).

4.1 Performance Comparison on the Haswell Microarchitecture

Fig. 9 shows the performance of the ChaCha20 encryption and decryption for a single Haswell core (Intel® Turbo Boost Technology, Intel® Hyper-Threading Technology, and Enhanced Intel Speedstep® Technology were disabled). The implementations were compiled with gcc version 4.8.1, AVX2 support ('-mavx2' – this also optimizes the SSE2/SSE3 [10] source code of the 128-bit vectorization with AVX [10] instructions) and compile time optimizations ('-O3' and '-fomit-frame-pointer'). We observe that the improvement for the 256-bit vectorization is only marginal for short messages of less than 128 bytes. However, for longer messages, our implementation almost doubles the performance, as shown in Fig. 9.

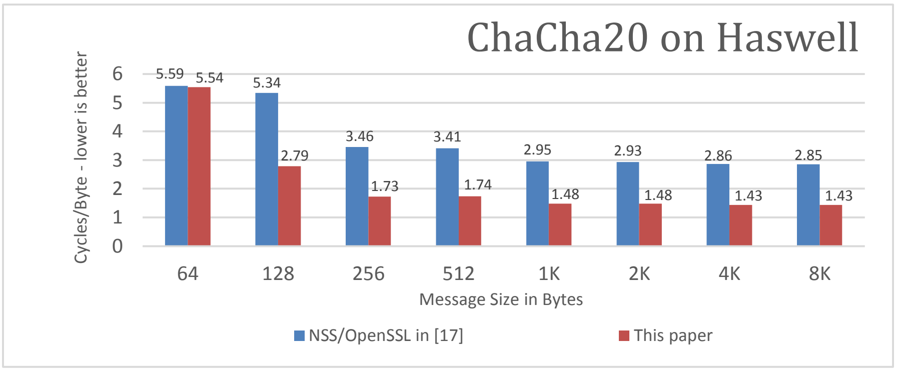

Fig. 9. Vectorized ChaCha20 encryption/decryption performance in cycles per byte for different message sizes on the Haswell microarchitecture.

4.2 Performance Estimation on Future Processors

A future Intel microarchitecture may introduce the AVX512 extension which could be used to further speed up ChaCha with the proposed 512-bit vectorization.

Since there is not yet any processor with the AVX512 extension available, we cannot measure the resulting performance at this point, and therefor use a different methodology. The compilation of the code can be done by using gcc version 4.9.0 (downloaded from Intel's website[1]), and the resulting binary can be executed on an emulator (Intel Software Development Emulator - SDE[1]). We used the SDE tool to count the number of executed instructions for the encryption and decryption. Table 1 compares the instructions count of the three implementations, for different message sizes. The first implementation is the 128-bit vectorization of the modified NSS and OpenSSL libraries; the second implementation is our 256-bit vectorization implementation and the third is our 512-bit vectorization implementation. The AVX512 based implementation offers no instruction overhead for smaller sized messages (less than 128 bytes) and it provides an incremental reduction for increasing message size from 16% to 56% in the instructions count. This indicates a high potential for a performance improvement on future processor generations.

Message size	Instructions count		
	NSS / OpenSSL, using AVX	This paper, using AVX2	This paper, using AVX512
64 B	499	504	423
128 B	927	496	422
256 B	1,745	905	459
512 B	3,409	1,758	811
1 KB	6,689	3,413	1,572
2 KB	13,297	6,750	3,023
4 KB	26,465	13,397	5,968
8 KB	52,849	26,718	11,815

Table 1. The instruction counts for the different vectorization implementations, using AVX, AVX2 and AVX512.

5 Discussion

We showed here how to significantly speed up the performance of ChaCha by means of widening the vectorization from 128-bit to 256-bit/512-bit and using algorithmic and software implementation improvements. Therefor we implemented the ChaCha algorithm using the new AVX2 extension feature of the Haswell microarchitecture and the future AVX512 extension feature. The evaluation shows for encrypting (or decrypting) more than 128 bytes, that the widening of the vectorization to 256-

[1] Available from http://software.intel.com/en-us/articles/intel-software-development-emulator

bit results in a doubling of the throughput for ChaCha. For less than 128 bytes the widened vectorization has no real effect. In addition, we could show with the instruction count comparison, that the 512-bit vectorization has a great capability to double again the throughput of ChaCha on future microarchitectures. These optimizations make ChaCha even more attractive for being a fast and secure alternative to AES in the TLS protocol.

References

1. Dierks, T., and Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC5246, IETF (2008), http://tools.ietf.org/html/rfc5246
2. Langley, A.: ChaCha20 and Poly1305 for TLS. In: Adam Langley's Weblog (October 2013), https://www.imperialviolet.org/2013/10/07/chacha20.html
3. AlFardan, N., Bernstein, D., Paterson, K., Poettering, B., and Schuldt, J.: On the security of RC4 in TLS and WPA. USENIX Security Symposium, 2013, https://www.usenix.org/conference/usenixsecurity13/security-rc4-tls
4. AlFardan, N., and Paterson, K.: Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. IEEE Symposium on Security and Privacy, 2013, http://www.isg.rhul.ac.uk/tls/TLStiming.pdf
5. Dworkin, M.: Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) for confidentiality and authentication. Federal Information Processing Standard Publication FIPS 800-38D, November, 2007, http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf
6. Gueron, S., and Krasnov, V.: Fast implementation of AES-CRT mode for AVX capable x86-64 processors. http://rt.openssl.org/Ticket/Display.html?id=3021&user=guest&pass=guest, March, 2013.
7. Bernstein, D. J.: ChaCha, a variant of Salsa20. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers, http://cr.yp.to/chacha/chacha-20080128.pdf
8. Bernstein, D. J.: The Poly1305-AES message-authentication code. Pages 32--49 in Fast software encryption: 12th international workshop, FSE 2005, http://cr.yp.to/mac/poly1305-20050329.pdf
9. Langley, A.: ChaCha20 and Poly1305 based Cipher Suites for TLS Draft 02. IETF DRAFT 02, IETF (2013), http://tools.ietf.org/html/draft-agl-tls-chacha20poly1305-02
10. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf (September 2013)
11. Intel: Intel® Architecture Instruction Set Extensions Programming Reference. http://download-software.intel.com/sites/default/files/319433-015.pdf (July 2013).
12. Bernstein, D. J.: The Salsa20 family of stream ciphers. Pages 84--97 in New stream cipher designs: the eSTREAM finalists, Springer (2008), http://cr.yp.to/snuffle/salsafamily-20071225.pdf
13. Bernstein, D. J: The ChaCha family of stream ciphers. In: D. J. Bernstein's webpage: http://cr.yp.to/chacha.html
14. ECRYPT Benchmarking of Stream Ciphers (eBACS) project webpage: http://bench.cr.yp.to/ebasc.html
15. Network Security Services (NSS) project webpage: https://developer.mozilla.org/en-US/docs/NSS
16. OpenSSL: The Open Source toolkit for SSL/TLS, project webpage:

http://www.openssl.org
17. The Chromium Projects, Open-source browser project webpage: https://sites.google.com/a/chromium.org/dev/Home
18. Chromium svn repository, http://src.chromium.org/svn/trunk/