White Paper

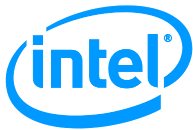
Jim Guilford
David Cote
Vinodh Gopal

IA Architects
Intel Corporation

Fast SHA512 Implementations on Intel® Architecture Processors

November 2012

328289-001



Executive Summary

The paper describes a family of highly-optimized implementations of the SHA512 cryptographic hash algorithm, which provide industry leading performance on a range of Intel® Processors for a single data buffer consisting of an arbitrary number of data blocks.

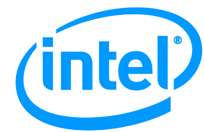
The paper describes the overall design of the SHA512 software, delves into some of the detailed optimizations, and presents a summary of the performance of some versions of the code. With our implementation, a single thread of an Intel® Core™ i7 processor 2600 can compute SHA512 of a large data buffer at the rate of ~8.59 cycles/byte[1].

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. www.intel.com/embedded/edc.

---

[1] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Configurations: Refer to the Performance section on page 13. For more information go to http://www.intel.com/performance.



Contents

Overview 4

Background of SHA512 4

Basic Design of Software 5

Software Versions 6

Message Scheduler Calculations 6

SIMD Optimizations 7

Performance 9

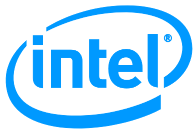
Methodology 9

Results 9

Conclusion 10

Contributors 11

References 11



Overview

This paper describes a family of highly-optimized implementations of the SHA512 cryptographic hash algorithm, which provide best performance on a range of Intel® processors for a single data buffer consisting of an arbitrary number of data blocks.

Background of SHA512

SHA512 [1] is one member of a family of cryptographic hash functions that together are known as SHA-2. The basic computation for the algorithm takes as input a block of input data that is 1024 bits (128 bytes) and a state vector that is 512 bits (64 bytes) in size, and it produces a modified state vector.

It is a follow-on to the earlier hash algorithms MD5 and SHA-1, and it is becoming increasingly important for secure internet traffic and other authentication problems. As the SHA512 processing involves a large amount of computations, it is critical that applications use the most efficient implementations available.

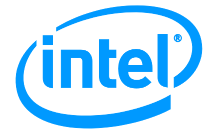
The algorithm operates on 64-bit QWORDs, so the state is viewed as 8 QWORDs (commonly called A…H) and the input data is viewed as 16 QWORDs.

The standard for the SHA-2 algorithm specifies a procedure for adding padding to the input data to make it an integral number of blocks in length. This happens at a higher level than the code described in this document. This paper is only concerned with updating the hash state values for any integral number of blocks.

The SHA512 algorithm is very similar to SHA256 [3], and most of the general optimization principles described in [3] apply here as well. The main differences in the algorithm specification are that SHA512 uses blocks, digests and data-type of computation twice the size of SHA256. In addition, SHA512 is specified with a larger number of rounds of processing (80 rather than 64).

The algorithm consists of two steps, as described in detail in [3]. The first step is a “message scheduler” that takes the input 16 QWORDs and computes 64 new QWORDs. Together with the original 16 QWORDs, these form a vector of 80 QWORDs that is the input to the second step.

This second step consists of 80 “rounds” where the form of the calculations in each round is the same. Each round takes as input the 8 state QWORDs, the corresponding input QWORD (after scheduling), and a round-specific constant, generating updated state QWORDS.



After all rounds have executed, the resulting state vector is added to the original state vector, and this results in the new state vector. If the input consists of multiple blocks, this process is repeated for each block.

Basic Design of Software

The nature of the round calculations is such that the rounds need to be processed in a serial manner. As a result, in most implementations this round calculation code executes completely on the scalar (or non-SIMD) execution unit of the processor. However, the messages scheduling calculations can be parallelized for a single block.

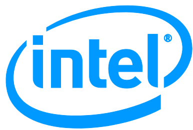
At a high level, our code is structured in this way: the message scheduling is done with SIMD instructions, whereas the rounds are done with scalar instructions. These two code sequences (the message scheduling and the rounds) are "stitched" together to further optimize performance. The basic idea of stitching is described in [4].

In particular, the message schedule will be calculated with SIMD instructions in pairs of QWORDs while scalar instructions compute the previous two rounds. The software pipelining method is summarized in the following table:

Iteration	SIMD Execution	Scalar Execution
1	Byteswap Message QWORDs 0 ... 1	
2	Byteswap Message QWORDs 2 ... 3	Rounds 0 ... 1
3	Byteswap Message QWORDs 4 ... 5	Rounds 2 ... 3
8	Byteswap Message QWORDs 14 ... 15	Rounds 12 ... 13
9	Compute Message QWORDs 16 ... 17	Rounds 14 ... 15
10	Compute Message QWORDs 18 ... 19	Rounds 16 ... 17
11	Compute Message QWORDs 20 ... 21	Rounds 18 ... 19
12	Compute Message QWORDs 22 ... 23	Rounds 20 ... 21
38	Compute Message QWORDs 74 ... 75	Rounds 72 ... 73
39	Compute Message QWORDs 76 ... 77	Rounds 74 ... 75
40	Compute Message QWORDs 78 ... 80	Rounds 76 ... 77
41		Rounds 78 ... 80

The first iteration computes the first pair of QWORDs of the message schedule. Iterations 2 through 40 compute pairs of QWORDs of the message schedule in parallel with 2 rounds. The last iteration only computes 2 rounds.

This allows for maximum parallelism while keeping the processing limited to a single data block.



Software Versions

The Intel® 64 instruction set architecture has SIMD instructions that include the Intel® SSE, SSE2 etc. extensions. The 2nd Generation Intel® Core™ processor family improves the performance of SIMD instructions with the introduction of the Intel® AVX1 (Intel® Advanced Vector Extensions) instruction set. The Intel® 64 processors built on 22nm process technology introduce rorx, which is an instruction set enhancement that allows non-destructive fast rotates by a constant, and 256-bit SIMD-integer instructions with Intel® AVX2.

Three versions of SHA512 code are available in [5] which we refer to as:

1. sha512_sse4
2. sha512_avx
3. sha512_avx2_rorx

The first uses the SSE instruction set for use on processors where the AVX1 instruction set is not present. The second uses AVX1 for use on processors that support AVX1, but where the rorx instruction [6] is not present. The third takes advantage of the rorx instruction and widened vectored integer operations in AVX2.

Message Scheduler Calculations

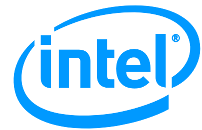
Let the scheduled message QWORDs be denoted by w[0]…w[79]. QWORDs w[0]…w[15] are computed by byte-swapping the input data due to the endianness of Intel® processors.

Each of the subsequent QWORDs w[i], where 16≤i<80, are formed by the following expressions.

s0[i] = (w[i-15] >>> 1) ⊕ (w[i-15] >>> 8) ⊕ (w[i-15] >> 7)
s1[i] = (w[i-2] >>> 19) ⊕ (w[i-2] >>> 61) ⊕ (w[i-2] >> 6)
w[i] = w[i-16] + w[i-7] + s0[i] + s1[i]

where
"⊕" indicates a bit-wise exclusive-or
">>>" indicates a right-rotate
">>" indicates a right-shift

Computation of w[i] depends upon the availability of w[i-16], w[i-15], w[i-7] and w[i-2]. Since w[i] has no dependency on w[i-1], w[i] and w[i-1] can be computed simultaneously. An XMM register can store a pair of QWORDs which allows the message schedule to be computed "2 at a time".



As mentioned previously, we compute the next pair of message schedule QWORDs while we are performing two rounds. In other words, the basic unit of work is to perform 2 rounds interleaved with computing one pair of message schedule QWORDs (one XMM register's worth). This is encoded in the SHA512_2Sched_2Round_SSE and SHA512_2Sched_2Round_AVX macros.

In the SSE and AVX versions of the code, a single input block is hashed in an unrolled loop structured like the following pseudo code.

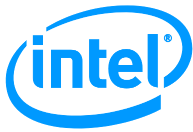
```
QWORD a,b,c,d,e,f,g,h; // State variables
Load State Variables from Digest;
For(i=0; i<41*2; i+=2){
	If(i<2){
		w[i, i+1] = Byteswap(Input[i, i+1]);
	}
	Elif(i<16){
		w[i, i+1] = Byteswap(Input[i, i+1]);
		SHA512_Round (i-2);
		SHA512_Round (i-1);
	}
	Elif (i<80){
		// Rounds i-2,i-1;
		// Compute w[i, i+1];
		SHA512_2Sched_2Round(i);
	}
	Else{
		SHA512_Round (i-2);
		SHA512_Round (i-1);
	}
}

Accumulate State Variables into Digest;
```

SIMD Optimizations

Stitching the message scheduling code with the round code and optimizations regarding the rotate operations on the A and E variable are similar to the techniques described in [3].

This same rotate optimization is applicable to the sigma functions in the message scheduler implementations.

s0(W[t-15])= (W[t-15]>>>1) ⊕ (W[t-15]>>>8) ⊕ (W[t-15]>>7)
s1(W[t-2])= (W[t-2]>>>19) ⊕ (W[t-2]>>>61) ⊕ (W[t-2]>>6)



Since there is no SIMD bitwise rotate instruction, the QWORD rotates must be computed using combinations of bitwise shift operations. Thus, it is necessary to first rewrite the expressions as follows.

$s0(W[t-15])=(W[t-15] \gg 1) \oplus (W[t-15] \ll 63) \oplus (W[t-15] \gg 8) \oplus (W[t-15] \ll 56) \oplus (W[t-15] \gg 7)$

$s1(W[t-2])=(W[t-2] \gg 19) \oplus (W[t-2] \ll 45) \oplus (W[t-2] \gg 61) \oplus (W[t-2] \ll 3) \oplus (W[t-2] \gg 6)$

The SIMD SSE instructions for computing parallel bitwise shifts on QWORDs are PSLLQ and PSRLQ. Because these instructions are destructive, implementing the above as written would involve a number of register copy operations. If, however, the expressions are rewritten as follows, the number of register copies can be minimized.

$s0(W[t-15])=((((W[t-15] \gg 1) \oplus W[t-15]) \gg 6) \oplus W[t-15]) \gg 1) \oplus ((W[t-15] \ll 7) \oplus W[t-15]) \ll 56$

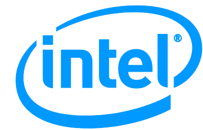
$s1(W[t-2])= ((((W[t-2] \gg 42) \oplus W[t-2]) \gg 13) \oplus W[t-2]) \gg 6) \oplus ((W[t-2] \ll 42) \oplus W[t-2]) \ll 3$

On AVX architectures, the VEX-encoded SIMD instructions are nondestructive. Therefore, there is no performance incentive to compute the sigma functions using this accumulate-shift technique if there are enough available XMM registers. Furthermore, separating the bitwise-shifts operations from the exclusive-or operations affords us more flexibility to reorder and interleave instructions.

One further optimization is to transfer the schedule QWORDs from the SIMD registers to the scalar execution unit through memory (that is, through a store and multiple loads), rather than directly moving them from register to register (using, for example, pextrd). The latter takes up an ALU slot, which is heavily in demand, whereas the load/store logic is relatively under-utilized at that time.

Each round calculation needs to add in the appropriate schedule QWORD along with a round-specific constant. The constant can be added to the schedule QWORD on the vector unit, before it is stored in memory for the scalar unit, for greater efficiency.

In each round calculation, all eight state variables must be rotated to their adjacent state variable. Rather than doing this rotation using register copy instructions, we rename the virtual registers (preprocessor symbols) to emulate the rotation. Thus, each round effectively rotates the set of state register names by one place. By doing an integer multiple of eight rounds in the body of the loop, the names rotate back to their starting values, so no register copies are needed before or after looping.



Performance

The performance results provided in this section were measured on widely available Intel® Processors. The SSE version was run on an Intel® Xeon® processor X5670, and the AVX1 version was run on an Intel® Core™ i7 processor 2600. In each case, the buffer size was swept in 128-byte increments. The tests were run with Intel® Turbo Boost Technology off.

Methodology

We measured the performance of the functions on data buffers of different sizes. We called the functions to hash the same buffer a large number of times, collecting many timing measurements. For each data buffer, we then sorted the timings, discarded the top and bottom 1/8th samples and then the largest/smallest quarter, and averaged the remaining quarter.

The timing was measured using the **rdtsc**() function which returns the processor time stamp counter (TSC). The TSC is the number of clock cycles since the last reset. The 'TSC_initial' is the TSC recorded before the function is called. After the function is complete, the **rdtsc**() was called again to record the new cycle count 'TSC_final'. The effective cycle count for the called routine is computed using

# of cycles = (TSC_final-TSC_initial).

A large number of such measurements were made for each data buffer and then averaged as described above to get the number of cycles for that buffer size. Finally, that value was divided by the buffer size to express the performance in cycles per byte.

Note: *Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.*

For more information go to http://www.intel.com/performance

Results

We show performance in cycles/byte for varying sizes of input data buffers, for various buffer sizes.

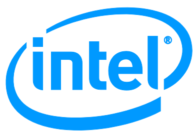
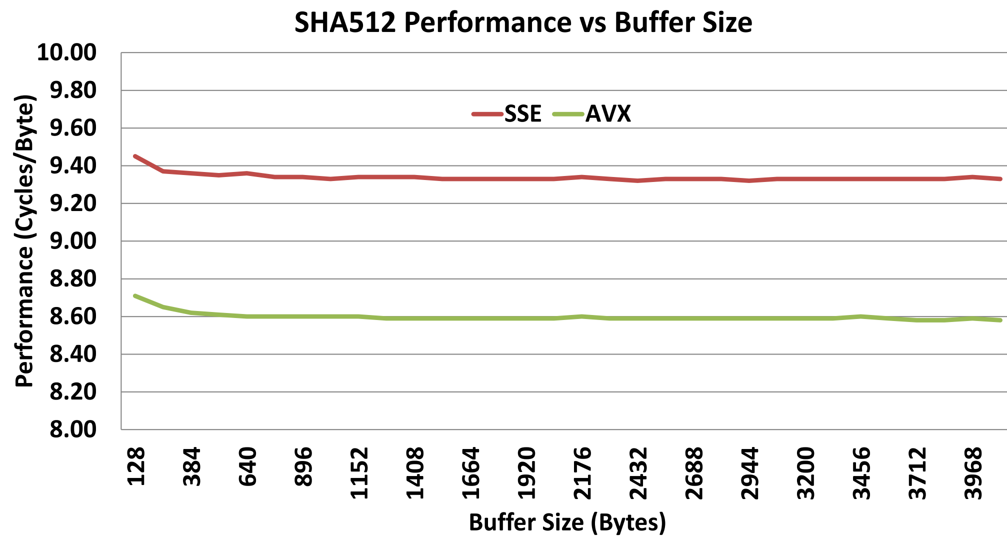


Figure 1: SHA512 Performance in Cycles/byte as a Function of Buffer Size (bytes)[2]



At the time of writing this paper, there are no widely available processors that support the rorx instruction.

The AVX code running on the Intel® Core™ i7 processor 2600, achieves a single-thread performance of 8.59 cycles/byte on large buffers.

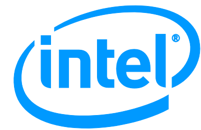
Conclusion

This paper presents three SHA512 implementations, optimized for different generations of Intel® processors. This is the fastest code for processing a single data buffer that we are aware of, that works on any number of blocks. We describe the high-level architecture of the code and a summary of some of the optimizations embedded in the code.

---

[2] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Configurations: Refer to the Performance section on page 13. For more information go to http://www.intel.com/performance.



Contributors

We thank Sean Gulley, Erdinc Ozturk, Kirk Yap and Wajdi Feghali for their substantial contributions to this work.

References

[1] "Federal Information Processing Standards Publication 180-2 SECURE HASH STANDARD" http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[2] "Processing Multiple Buffers in Parallel to Increase Performance on Intel® Architecture Processors"

[3] Fast SHA256 Implementations on Intel® Architecture Processors http://download.intel.com/embedded/processor/whitepaper/327457.pdf

[4] "Fast Cryptographic Computation on Intel® Architecture Processors Via Function Stitching".

[5] Optimized SHA512 Source Code http://www.intel.com/p/en_US/embedded/hwsw/technology/packet-processing#docs

[6] Intel® Advanced Vector Extensions Programming Reference http://software.intel.com/file/36945

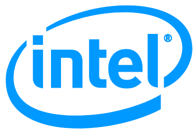
The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://intel.com/embedded/edc.

Authors

Jim Guilford, David Cote, Vinodh Gopal are IA Architects with the IAG Group at Intel Corporation.

Acronyms

IA Intel® Architecture



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order. Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Hyper-Threading Technology requires a computer system with a processor supporting HT Technology and an HT Technology-enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. For more information including details on which processors support HT Technology, see here.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

Intel® Turbo Boost Technology requires a PC with a processor with Intel Turbo Boost Technology capability. Intel Turbo Boost Technology performance varies depending on hardware, software and overall system configuration. Check with your PC manufacturer on whether your system delivers Intel Turbo Boost Technology.For more information, see http://www.intel.com/technology/turboboost.

Intel, Intel Turbo Boost Technology, Intel Hyper Threading Technology, Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2012 Intel Corporation. All rights reserved.