White Paper

Jim Guilford
Kirk Yap
Vinodh Gopal

IA Architects
Intel Corporation

Fast SHA-256 Implementations on Intel® Architecture Processors

May 2012

327457-001

Executive Summary

The paper describes a family of highly-optimized implementations of the Fast SHA-256 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 Fast SHA-256 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 core of an Intel® Core™ i7 processor 2600 with Intel® HT Technology can compute Fast SHA-256 of a large data buffer at the rate of ~11.5 cycles/byte[1].

[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

Contents

Overview

This paper describes a family of highly-optimized implementations of the Fast SHA-256 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.

Background of Fast SHA-256

Fast SHA-256 [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 a block of input data that is 512 bits (64 bytes) and a state vector that is 256 bits (32 bytes) in size, and it produces a modified state vector.

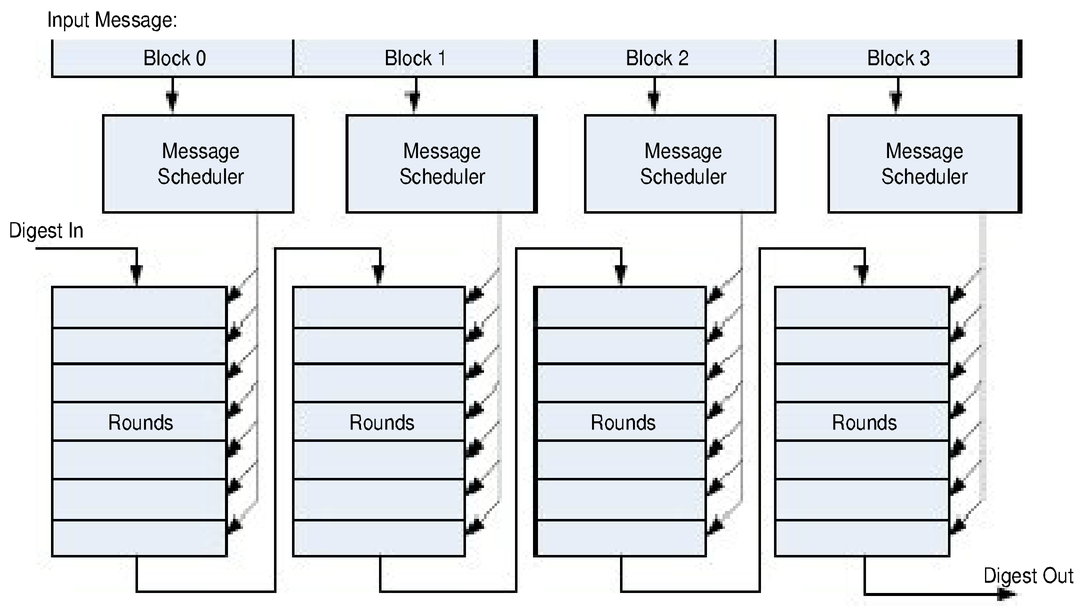
It is a follow-on to the earlier hash algorithms MD5 and SHA-1, and it is becoming increasingly important for secure internet traffic and other authentication problems. As the Fast SHA-256 processing involves a large amount of computations, it is critical that applications use the most efficient implementations available.

The algorithm operates on 32-bit DWORDs, so the state is viewed as 8 DWORDs (commonly called A…H) and the input data is viewed as 16 DWORDs.

The standard for the SHA-2 algorithm specifies a procedure for adding padding to the input data to make it an integral number of (64-byte) 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 algorithm can viewed as consisting of two steps, as shown in the following figure.

Figure 1: Steps in Fast SHA-256 Algorithm processing



The first step is a "message scheduler" that takes the input 16 DWORDs and computes 48 new DWORDs. Together with the original 16 DWORDs, these form a vector of 64 DWORDs that is the input to the second step.

This second step consists of 64 "rounds" where the form of the calculations in each round is the same. Each round takes as input the 8 state DWORDs, the corresponding input DWORD (after scheduling), and a round-specific constant.

After all 64 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. That is, one cannot perform the calculations for round *i*, until the calculations for round *i-1* are completed. As a result, in most implementations this round calculation code executes completely on the integer (or non-SIMD) execution unit of the processor.

One variation is a Single Instruction Multiple Data (SIMD) implementation of the integer version, which will hash data from multiple independent streams at the same time. This basic approach is described in [2]. This gives better performance, but in some applications there are not multiple independent streams available. That is, the application may want to compute the hash of some buffer of data, and it will not have a different buffer available until after the hash of the first buffer has already been computed. In this case, one needs to find the fastest way to compute the hash of a single buffer.

As stated, the round calculations are inherently serial and cannot be parallelized (for a single input stream). However, the messages scheduling calculations can be parallelized in at least two different ways. One can schedule the data associated with a single block in a parallel manner, one can schedule multiple blocks (for the same data stream) in parallel, or a combination of these. The technique of scheduling multiple blocks in parallel is proposed and described in more detail in [3].

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 integer 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 first 16 message DWORDs are available immediately (they consist of the actual input DWORDs). So while the first 16 rounds are being computed, message DWORDs (16…31) can be computed, etc. This is shown below:

Integer Execution	**SIMD Execution**
Execute Rounds 0...15	Compute Message DWORDs 16…31
Execute Rounds 16...31	Compute Message DWORDs 32…47
Execute Rounds 32...47	Compute Message DWORDs 48…63
Execute Rounds 48...63	

For the first 48 rounds, the code performs both round calculations and message scheduling, but for the last 16 rounds no scheduling is done.

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

Software Versions

Four versions of Fast SHA-256 code are available in [5] which we refer to as:

1. sha256_sse4
2. sha256_avx1
3. sha256_avx2_rorx2
4. sha256_avx2_rorx8

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 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.

Two versions use the rorx instruction: sha256_avx2_rorx2, and sha256_avx2_rorx8. The former is optimized for smaller buffers, a smaller memory foot-print, and single-thread execution. The latter is optimized for larger buffers, particularly when Intel® Hyper-Threading Technology (Intel® HT Technology) is enabled, but it has a larger memory footprint, which might result in worse data-cache behavior.

Message Scheduler Calculations

Let the scheduled message DWORDs be denoted by w[0]…w[63]. Then w[0]…w[15] are formed by the input data itself. Each of the subsequent DWORDs w[i], where $(16 \le i < 64)$, is formed by the expressions:

$$
\begin{aligned}
s0[i] &= (w[i-15] \mathbin{>>>} 7) \oplus (w[i-15] \mathbin{>>>} 18) \oplus (w[i-15] \mathbin{>>} 3)\\
s1[i] &= (w[i-2] \mathbin{>>>} 17) \oplus (w[i-2] \mathbin{>>>} 19) \oplus (w[i-2] \mathbin{>>} 10)\\
w[i] &= w[i-16] + w[i-7] + s0[i] + s1[i]
\end{aligned}
$$

where
"$\oplus$" is a bit-wise XOR
">>>" indicates a right-rotate
">>" indicates a right-shift

Note that the furthest reference for computing w[i] is w[i-16], so in general we only need to store 16 of the DWORDs at any time.

We can store 4 DWORDs in a single XMM register, so we can store the necessary scheduled message DWORDs in four such registers:

XMM	DWORD			
Register	3	2	1	0
X3	w[i-1]	w[i-2]	w[i-3]	w[i-4]
X2	w[i-5]	w[i-6]	w[i-7]	w[i-8]
X1	w[i-9]	w[i-10]	w[i-11]	w[i-12]
X0	w[i-13]	w[i-14]	w[i-15]	w[i-16]

From this, we want to compute w[i+3]…w[i+0]. We can compute the following "4 at a time", because this sub-expression only depends on data that is in these four registers:

$$
\begin{aligned}
s0[i] &= (w[i-15] \mathbin{>>>} 7) \oplus (w[i-15] \mathbin{>>>} 18) \oplus (w[i-15] \mathbin{>>} 3)\\
w'[i] &= w[i-16] + w[i-7] + s0[i]
\end{aligned}
$$

Unfortunately, this does not extend to s1[], since s1[i+3] depends on w[i+1], which has not yet been calculated. So the s1[] calculations need to be "2 at a time", which means that they need to be done twice. That is, we do one set of calculations to compute {s1[i+1], s1[i]}, and then we do a second set of calculations to compute {s1[i+3], s1[i+2]}.

As mentioned previously, we compute the next set of 16 scheduled message DWORDs while we are processing the current set. More specifically, the basic unit of work is to perform 4 rounds interleaved with computing one set of 4 new scheduled message DWORDs (one XMM register's worth). This is encoded in the FOUR_ROUNDS_AND_SCHED macro.

In the SSE and AVX1 versions of the code, the basic loop body contains four invocations of this macro, so it performs 16 rounds and computes 16 new DWORDs. This loop repeats for 3 iterations, and then a loop containing 8 rounds only is repeated twice. Note that both the SSE and AVX1 versions make use of 128-bit SIMD-integer instructions for the scheduling.

The rorx versions are similar, except that we can also use the AVX2 instruction extensions which support SIMD-integer instructions that work on registers twice as wide (256-bit). This means that each YMM register can hold two blocks of data (in the same format as described above). To take advantage of this, we process two blocks at a time. The first block is processed in a similar manner to the AVX1 version, except that in parallel with scheduling the data for the first block, the scheduling for the second block is also done, with the results written to the stack.

When the rounds for the first block are finished, the rounds for the second block are executed with no scheduling, using the previously computed scheduled DWORDs.

If there is an odd number of blocks, then for the last block the second set of rounds are skipped.

So for sha256_avx2_rorx2, the scheduling is done two blocks at a time, with four (DWORD) lanes used per block.

On the other hand, the sha256_avx2_rorx8 code schedules eight blocks at a time (using only one lane per block). It is then followed by the round calculations for seven blocks with no scheduling. This imposes extra overhead when the number of blocks is not a multiple of 8. For larger buffers (roughly more than 1 KByte), the efficiency for the bulk outweighs the inefficiency for the remainder, but for smaller buffers (particularly for 1 to 7 blocks), the overhead is greater.

Scheduling more than one block at a time was proposed in [3].

SSE/AVX1 Version Optimizations

Stitching the message scheduling code with the round code gives an obvious advantage in that it reduces inter-instruction dependencies, resulting in more efficient use of processor execution resources. It also provides a less obvious advantage.

Particularly on the non-AVX processors, the front-end decoders may become the bottle-neck, limiting total performance. By having two independent instructions streams that can be mixed arbitrarily, we have much greater flexibility to re-order the instruction sequence to one that will optimize the front-end decoder's performance.

While computing the s1 function on the vector unit "by-2" is less efficient than computing it 4 at a time, this does offer one optimization. A limitation of the SSE/AVX instruction set is that there are no vector rotate instructions. This means that doing a rotate on each lane needs to be done using two shifts and an OR (or XOR).

However, since for s1 we are only using half of the available lanes, we can duplicate the data into two adjacent lanes. For example, if we wanted to operate on DWORDs A and B, we would load the four lanes of the register with {B, B, A, A}. We can then accomplish a DWORD rotation by doing a single QWORD shift (resulting in two lanes containing the correct DWORD data, and two lanes containing garbage).

One optimization in the round's calculation is in calculating the sigma functions on the a and e state variables:

$$S0(a) = (a >>> 2) \oplus (a >>> 13) \oplus (a >>> 22)$$
$$S1(e) = (e >>> 6) \oplus (e >>> 11) \oplus (e >>> 25)$$

Because the ror instruction is destructive (that is, it overwrites the source operand), implementing the above as written would involve a number of register copy operations. If, however, the expressions are rewritten as:

$$S0(a) = (((a >>> 9) \oplus a) >>> 11) \oplus a) >>> 2$$
$$S1(e) = (((e >>> 14) \oplus e) >>> 5) \oplus e) >>> 6$$

Then the number of register copies can be minimized. Both S0 and S1 are computed at the same time (in an interleaved manner) to avoid serializing the chain of dependent rotates.

One further optimization is to transfer the scheduled DWORDs from the SIMD registers to the integer 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.

In each round calculation six out of the eight state variables are shifted to the next state variable. Rather than do these using mov instructions, we rename the virtual registers (symbols) to effect this “shift”. Thus each round effectively rotates the set of state register names by one place. By doing 8 or 16 rounds in the body of the loop, the names have rotated back to their starting values, so no register moves are needed before looping.

Similarly on the vector unit for the message scheduling, the 16 necessary scheduled DWORDs are stored in four XMM registers, as described earlier. For example, the initial data DWORDs are stored in order as {X0, X1, X2, X3}. When we compute four new scheduled DWORDs, we store them in X0 (overwriting the “oldest” data DWORDs), so now the scheduled DWORDs are stored in order in {X1, X2, X3, X0}. Once again, we handle this by “rotating” the four names, where in this case the names rotate one place every four rounds (because we compute four scheduled DWORDs in each calculation). By having 16 rounds (four scheduling operations) in the body of the loop, these XMM register names rotate back to their initial value, and again no register moves are needed before looping.

Note that in the last 16 rounds, no scheduling takes place, so we can get by with 8 rounds rather than 16 in the body of the loop.

Each round calculation needs to add in the appropriate scheduled DWORD along with a round-specific constant. The constant can be added to the scheduled DWORD on the vector unit, before it is sent to the integer unit, for greater efficiency.

One last optimization is that since we are storing 4 DWORDs in each of 4 XMM registers, we can load one block in four loads rather than sixteen. Similarly, we can byte-swap these in four rather than sixteen operations.

Optimizations with rorx

There are several differences in the rorx versions. The most notable, as described earlier, is the use of YMM registers to schedule multiple blocks in parallel.

But there are also changes in the details of the round calculations. The ror instruction is destructive, meaning that the output overwrites the input. Thus, to minimize register copy operations, we re-factored the S0 and S1 calculations as described earlier. However the rorx instruction is non-destructive, in that it has separate source and destination arguments. This allows the code to be rewritten in a way that more efficiently executes with a shorter critical path.

The round Sigma functions, S0 and S1 include three rotates each and benefit significantly with this enhancement.

The calculation of S1 is critical in the performance optimization of the round function. S1 is used to calculate both the new E and new A states. Completing the calculation of S1 early is therefore very significant. With the introduction of rorx, which results in faster rotates, we are able to complete the calculation of S1 much earlier by calculating the rotates in parallel. In the SSE/AVX1 versions of the code, the three rotates in the calculation of S1 are performed in a sequential, dependent manner. Instead we perform two of the three rotates in parallel with the use of only one extra temporary register. The non-destructive nature of rorx allows us to avoid additional moves.

Rorx-based Code sequence:

```
rorx  y0, e, 25   ; y0 = e >> 25                 ; S1A
rorx  y1, e, 11   ; y1 = e >> 11                 ; S1B
rorx  T1, a, 13   ; T1 = a >> 13                 ; S0A
xor   y0, y1      ; y0 = (e>>25) ⊕ (e>>11)       ; S1
rorx  y1, e, 6    ; y1 = (e >> 6)                ; S1
xor   y0, y1      ; y0 = S1 = (e>>25) ⊕ (e>>11) ⊕ (e>>6) ; S1
rorx  y1, a, 22   ; y1 = a >> 22                 ; S0B
xor   y1, T1      ; y1 = (a>>22) ⊕ (a>>13)       ; S0
rorx  T1, a, 2    ; T1 = (a >> 2)                ; S0
xor   y1, T1      ; y1 = S0 = (a>>22) ⊕ (a>>13) ⊕ (a>>2) ; S0
```

rorx8 Version Optimizations

The most efficient way to read 8 blocks of input data is to read them in 8 DWORDs of each block into each of the 8 YMM registers and "transpose them", before writing them to the stack. This is then repeated with the second set of 8 DWORDs. This is significantly faster than reading the data DWORD by DWORD, and it is faster than using the gather instruction.

Unfortunately, since there are not enough architectural registers to store 16 DWORDs for 8 blocks in registers, the scheduled DWORDs are stored in an array on the stack. Since most of the scheduled DWORDs are not used until subsequent blocks are processed, all 64 scheduled DWORDs need to be stored, not just 16. This means that this version of the code uses significantly more stack space than the other versions.

There are two large arrays: TMSG and KTMSG. The former holds the scheduled DWORDS, and the latter holds the scheduled DWORDS with the round constant added. The latter is needed for the rounds, whereas the former is needed during scheduling.

One way to reduce the memory footprint would be to change the scheduler to add the constant to DWORD *i-16* after the scheduling code has read that DWORD and before the stitched round code has read it. Unfortunately, the round code reads that DWORD fairly early, and the scheduling code reads that DWORD fairly late. The code could be rearranged to avoid this but performance slowed down by a few percent.

Instead, we overlap the two arrays, so that TMSG[0] is at the same address as KTMSG[16]. This means that when we write KTMSG[*i*], we clobber TMSG[*i-16*]. This is functionally correct because once we have computed scheduled DWORD *i*, we no longer need TMSG[*i-16*] or earlier.

Conceptually, this is similar to have a 64-element array for KTMSG and a 16-element array for TMSG. We cannot do that directly as we use the same index into both arrays, but the trick described above allows us to have the same effect and still use the same index.

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 64-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.

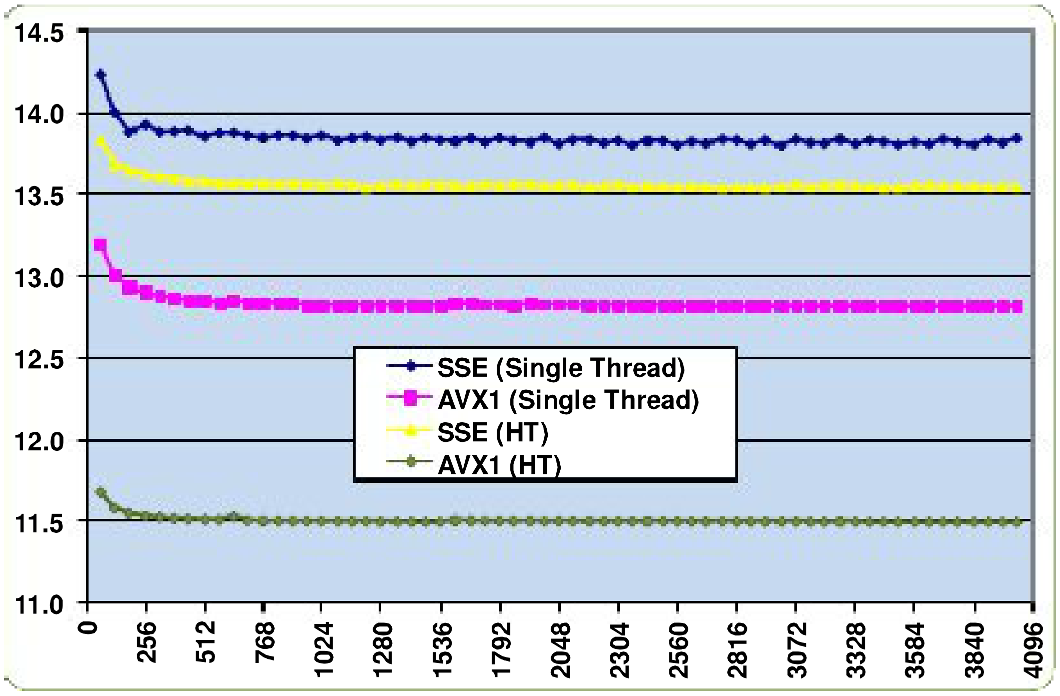
Note: *Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors 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 2: Fast SHA-256 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.

[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

Conclusion

This paper presents four Fast SHA-256 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.

Contributors

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

References

2.pdf

[3] S. Gueron, V. Krasnov: "Parallelizing message schedules to accelerate the computations of hash functions" http://eprint.iacr.org/2012/067.pdf

Authors

Jim Guilford, Kirk Yap, and Vinodh Gopal are IA Architects with the IAG Group at Intel Corporation.

Acronyms

IA	Intel® Architecture