BLAKE and 256-bit advanced vector extensions

Samuel Neves[1] and Jean-Philippe Aumasson[2]

[1] University of Coimbra, Portugal
[2] NAGRA, Switzerland

Abstract. Intel recently documented its AVX2 instruction set extension that introduces support for 256-bit wide single-instruction multiple-data (SIMD) integer arithmetic over double (32-bit) and quad (64-bit) words. This will enable Intel's future processors—starting with the Haswell architecture, to be released in 2013—to fully support 4-way SIMD computation of 64-bit ARX algorithms (32-bit is already supported since SSE2). AVX2 also introduces instructions with potential to speed-up cryptographic functions, like any-to-any permute and vectorized table lookup. In this paper we show how the AVX2 instructions will benefit the SHA-3 finalist hash function BLAKE, an algorithm that naturally lends itself to 4-way 32- or 64-bit SIMD implementations thanks to its inherent parallelism. We also wrote BLAKE-256 assembly code for AVX and AVX2, and measured for the former a speed of 7.62 cycles per byte, setting a new speed record.

Keywords: hash functions, SHA-3, implementation, SIMD

1 Introduction

NIST will announce the winner of the SHA-3 competition[3] in the second quarter of 2012. At the time of writing (February 26, 2012), no significant security weakness has been discovered on any of the five finalists—BLAKE, Grøstl, JH, Keccak, and Skein—and all seem to provide a comfortable margin of security against future attacks. It is thus expected that secondary evaluation criteria such as performance and ease of implementation will be decisive in the choice of SHA-3.

An important secondary criterion is the hashing speed on high-end CPUs, such as the popular Core-branded Intel processors. These processors are found in laptops, desktops, and servers, and it is likely that users hashing large amounts of data (be it many small messages or a few big ones) will only switch from SHA-2 to SHA-3 if the latter is noticeably faster. An example of application where fast hashing is needed is the authentication of data in secure cloud storage services. The relative hashing speeds of the SHA-3 finalists are found on the dedicated page of the eBACS [1] project[4]: `http://bench.cr.yp.to/results-sha3.html`.

This paper focuses on the SHA-3 finalist BLAKE, and investigates implementations exploiting Intel's upcoming AVX2 instruction set [2]. Indeed, previous implementations of BLAKE have exploited the SSE2, SSSE3, and SSE4.1 instruction sets, which provide single-input multiple-data (SIMD) instructions over 128-bit XMM registers. Thanks to BLAKE's inherent internal parallelism, such instructions generally lead to a significant speed-up. AVX2 extends SIMD capabilities to 256-bit registers, and thus provides a new avenue for speed-optimized implementations of BLAKE.

We wrote complete C and assembly implementations of BLAKE-512 for AVX2. As AVX2 is not rolled out in today's CPU's, a best effort was to make heuristical estimates based on the information available. We also wrote assembly implementations of BLAKE-256 for AVX and AVX2, and measured the former on a Sandy Bridge CPU at 7.62 cycles per byte, improving on the 7.87 figure from eBASH (`sandy0`, `supercop-20110708`). We used Intel's Software Development

[3] `http://www.nist.gov/hash-competition`.

[4] The eBACS project relies on the SUPERCOP benchmark toolkit, to performs automated benchmarks of a number of primitives on various software platforms, testing each implementation submitted with a number of compilation options, and reporting the fastest combination.

Emulator[5] to test the correctness of the AVX2 implementations, and the latest trunk build of the Yasm assembler[6] (as the latest release does not support AVX2) to compile them.

2 The keyed permutation of BLAKE

The SHA-3 finalist BLAKE is composed of two main hash functions, BLAKE-256 and BLAKE-512. Below we only describe the keyed permutation algorithms at the core of their respective compression functions, for it is the only performance-critical part in a software implementation. We refer to [3] for a complete specification of BLAKE.

The keyed permutations of both BLAKE-256 and BLAKE-512 transform 16 words $v_0, v_1, \dots, v_{15}$ and use

- 16 message words $m_0, m_1, \dots, m_{15}$ as the key of the permutation,
- 16 word constants $u_0, u_1 \dots, u_{15}$,
- ten permutations of the set $\{0, \dots, 15\}$ denoted $\sigma_0, \sigma_1, \dots, \sigma_{15}$.

BLAKE-256 operates on 32-bit words and BLAKE-512 operates on 64-bit words. The u constants are thus different for the two functions.

A round of the keyed permutation makes two layers of computations using the G function, respectively on the columns and on the diagonals of the 4×4 array representation of $v_0, v_1, \dots, v_{15}$, as described below:

$$\begin{array}{llll}
\mathsf{G}_0(v_0\ ,v_4\ ,v_8\ ,v_{12}) & \mathsf{G}_1(v_1\ ,v_5\ ,v_9\ ,v_{13}) & \mathsf{G}_2(v_2\ ,v_6\ ,v_{10},v_{14}) & \mathsf{G}_3(v_3\ ,v_7\ ,v_{11},v_{15}) \\
\mathsf{G}_4(v_0\ ,v_5\ ,v_{10},v_{15}) & \mathsf{G}_5(v_1\ ,v_6\ ,v_{11},v_{12}) & \mathsf{G}_6(v_2\ ,v_7\ ,v_8\ ,v_{13}) & \mathsf{G}_7(v_3\ ,v_4\ ,v_9\ ,v_{14})
\end{array}$$

The bijective transform $\mathsf{G}_i(a, b, c, d)$ does the following operations:

$$\begin{aligned}
&a \leftarrow a + b + (m_{\sigma_r[2i]} \oplus u_{\sigma_r[2i+1]}) \\
&d \leftarrow (d \oplus a) \ggg \alpha \\
&c \leftarrow c + d \\
&b \leftarrow (b \oplus c) \ggg \beta \\
&a \leftarrow a + b + (m_{\sigma_r[2i+1]} \oplus u_{\sigma_r[2i]}) \\
&d \leftarrow (d \oplus a) \ggg \gamma \\
&c \leftarrow c + d \\
&b \leftarrow (b \oplus c) \ggg \delta
\end{aligned}$$

where the round index $r \geq 0$ is reduced modulo 10. BLAKE-256 operates on 32-bit words, performs 14 rounds, and uses rotation constants $\alpha = 16$, $\beta = 12$, $\gamma = 8$, and $\delta = 7$. BLAKE-512 operates on 64-bit words, performs 16 rounds, and uses rotation constants $\alpha = 32$, $\beta = 25$, $\gamma = 16$, and $\delta = 11$.

The four G functions of the first layer (called "column step" in [3]) can be computed in parallel, as well as the four of the second layer (called "diagonal step"). One can thus view a round as

1. a column step;
2. a left-rotation of the i-th column by i positions, $i = 0, 1, 2, 3$, i.e.,

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \text{ is transformed to } \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_5 & v_6 & v_7 & v_4 \\ v_{10} & v_{11} & v_8 & v_9 \\ v_{15} & v_{12} & v_{13} & v_{14} \end{pmatrix} \ ;$$

[5] `http://software.intel.com/en-us/articles/intel-software-development-emulator/`

[6] `http://yasm.tortall.net/`

3. a column step;
4. a right-rotation of the i-th column by i positions, $i = 0, 1, 2, 3$, to reset words to their initial position.

These two observations make a 4-way word-vectorized implementation of a round straightforward, as shown in the next section.

3 Previous SIMD implementations of BLAKE

A number of previous implementations of BLAKE (as included SUPERCOP [1]) have used Intel's Streaming SIMD Extensions (SSE) instruction sets to exploit the parallelism of BLAKE for improved performance. This section gives a brief overview of those implementations, starting with a presentation of the SSE2 instruction set.

3.1 Streaming SIMD Extensions 2 (SSE2)

Intel's first set of instructions supporting all 4-way 32-bit SIMD operations necessary to implement BLAKE-256 is the Streaming SIMD Extensions 2 (SSE2) set. SSE2 includes vector instructions on 4×32-bit words for integer addition, XOR, word-wise left and right shift, as well as word shuffle. This is all one needs to implement BLAKE-256's round function, as rotations can be simulated by two shifts and an XOR. BLAKE-512 can also use SSE2 (though with less benefit than BLAKE-256), thanks to the support of 2-way 64-bit SIMD operations.

SSE2 instructions operate on 128-bit XMM registers, rather than 32 or 64-bit general-purpose registers. In 64-bit mode (i.e. in `x86-64` a/k/a `amd64` architecture) 16 XMM registers are available, whereas only eight are available in 32-bit mode. The SSE2 instructions are supported by all recent Intel and AMD desktop and laptop processors (Intel's Xeon, Celeron, Core X's, etc.; AMD's Athlon 64, Opteron, etc.) as well as by common low-voltage processors, as found in netbooks (Intel's Atom; VIA's C7 and Nano).

In addition to inline assembly, C(++) programmers can use SSE2 instructions via *intrinsic functions* (or intrinsics), which are API extensions built into the compiler. Intrinsics allow to enforce the use of SSE2 instructions by the processor, enable the use of C syntax and variables instead of assembly language and hardware registers, and let the compiler optimize instruction scheduling for better performance. Table 1 shows intrinsics corresponding to some assembly mnemonics used to implement BLAKE-256. A complete reference to SSE2 intrinsics can be found in [4].

Table 1. Intrinsics of main SSE2 instructions used to implement BLAKE-256.

Assembly	Intrinsic	Description
`paddd`	`_mm_add_epi32`	4-way 32-bit integer addition
`pshufd`	`_mm_shuffle_epi32`	4-way 32-bit word shuffle
`pslld`	`_mm_srli_epi32`	4-way 32-bit left-shift
`psrld`	`_mm_srli_epi32`	4-way 32-bit right-shift
`pxor`	`_mm_xor_si128`	Bitwise XOR of two 128-bit registers

3.2 SIMD implementation of BLAKE-256 using SSE2

To help understand the principle of SIMD implementations of BLAKE, we first present a simple SSE2 implementation of BLAKE-256's column step, similar to the `sse2` implementation in SUPERCOP. The v internal state is stored in four XMM registers defined as `__m128i` type and aliased `row1`, `row2`, `row3`, and `row4`. These respectively correspond to the first four rows of the 4×4 array representation described in §2.

First, one initializes a 128-bit XMM register aliased `buf1` with the four message words $m_{\sigma_r[2i]}$. Another XMM register aliased `buf2` is initialized with the four constants $u_{\sigma_r[2i+1]}$. `buf1` and `buf2` are XORed together into `buf1` and the result is added to `row1`:

```
buf1 = _mm_set_epi32(m[sig[r][6]], m[sig[r][4]],
                     m[sig[r][2]], m[sig[r][0]]);
buf2 = _mm_set_epi32(u[sig[r][7]], u[sig[r][5]],
                     u[sig[r][3]], u[sig[r][1]]);
buf1 = _mm_xor_si128( buf1, buf2 );
row1 = _mm_add_epi32( row1, buf1 );
```

At this state, one can already prepare the XMM register containing the XOR of the permuted message and constants for the next message input:

```
buf1 = _mm_set_epi32(m[sig[r][7]], m[sig[r][5]],
                     m[sig[r][3]], m[sig[r][1]]);
buf2 = _mm_set_epi32(u[sig[r][6]], u[sig[r][4]],
                     u[sig[r][2]], u[sig[r][0]]);
buf1 = _mm_xor_si128( buf1, buf2);
```

The subsequent operations are only vectorized XOR, integer addition, and word-wise shifts:

```
row1 = _mm_add_epi32( row1, row2 );
row4 = _mm_xor_si128( row4, row1 );
row4 = _mm_xor_si128( _mm_srli_epi32( row4, 16 ),
                      _mm_slli_epi32( row4, 16 ));
row3 = _mm_add_epi32( row3, row4 );
row2 = _mm_xor_si128( row2, row3 );
row2 = _mm_xor_si128( _mm_srli_epi32( row2, 12 ),
                      _mm_slli_epi32( row2, 20 ));
row1 = _mm_add_epi32( row1, buf1 );
row1 = _mm_add_epi32( row1, row2 );
row4 = _mm_xor_si128( row4, row1 );
row4 = _mm_xor_si128( _mm_srli_epi32( row4,  8 ),
                      _mm_slli_epi32( row4, 24 ));
row3 = _mm_add_epi32( row3, row4 );
row2 = _mm_xor_si128( row2, row3 );
row2 = _mm_xor_si128( _mm_srli_epi32( row2,  7 ),
                      _mm_slli_epi32( row2, 25 ));
```

At the end of a column step, each register is word-rotated to perform the diagonal step as a column step on the rotated state, as observed in §2:

```
row2 = _mm_shuffle_epi32( row2, _MM_SHUFFLE(0,3,2,1) );
row3 = _mm_shuffle_epi32( row3, _MM_SHUFFLE(1,0,3,2) );
row4 = _mm_shuffle_epi32( row4, _MM_SHUFFLE(2,1,0,3) );
```

The `_mm_shuffle_epi32` intrinsic takes as second argument an immediate value (i.e. a constant integer literal) expressed as a predefined macro. We refer to [4, p.65] for details of the `_MM_SHUFFLE` macro.

3.3 Implementations using SSSE3 and SSE4.1

The SSE2 instruction set was followed by the SSE3, SSSE3, SSE4.1 and SSE4.2 extensions [4], which brought additional instructions to operate on XMM registers. It was found that some of those instructions could be of benefit to BLAKE, and implementations exploiting SSSE3 and SSE4.1 instructions have been submitted to SUPERCOP:

- The `ssse3` implementation of BLAKE-256 uses the `pshufb` instruction (intrinsic `_mm_shuffle_epi8`) to perform rotations of 16 and 8 bits, as well as the initial conversion of the message from little-endian to big-endian byte order, since both can be expressed as byte shuffles (in the sse2 implementations rotations were implemented as two shifts and an XOR). This brings a significant speed-up on Core 2 based on the Penryn microarchitecture, which introduced a dedicated shuffle unit to complete pshufb within one micro-operation, against four on the first Core 2 chips [5].

- The sse41 implementation of BLAKE-256 uses the pblendw instruction (_mm_blend_epi16) in combination with SSE2's pshufd, pslldq, and others to load m and u words according to the σ permutations without using table lookups.

In general, the ssse3 implementation is faster than sse2, and sse41 is faster than both[7]. For example, the 20110708 measurements of SUPERCOP on sandy0 (a machine equipped with a Sandy Bridge Core i7, without AVX activated) report sse41 as the fastest implementation of BLAKE-256, with the ssse3 and sse2 implementations being respectively 4 % and 24 % slower.

Recently, SUPERCOP included the vect128 and vect128-mmxhack implementations of BLAKE-256 by Leurent, which slightly outperform the sse41 implementation. The main singularity of Leurent's code is its implementation of the σ permutations: vect128 "byte-slices" each message word accross four XMM registers and uses the pshufb instruction to reorder them according to σ; vect128-mmxhack instead uses MMX and general-purpose registers to store and unpack the message words in the correct order into XMM registers.

4 AVX2 advanced vector extensions

AVX2 is an extension of Intel's Advanced Vector Extensions (AVX) set of instructions, somewhat in the same spirit of SSE2's extension of SSE: AVX and SSE do not provide integer vector operations, while AVX2 and SSE2 do. This section gives a brief overview of AVX and AVX2, and describes in more detail the instructions that will be used to implement BLAKE. We refer to Intel's reference documents for a complete documentation of AVX [6] and AVX2 [2].

4.1 Overview of AVX and AVX2

Advanced Vector Extensions (AVX) were announced by Intel in 2008 to introduce 256-bit wide vector instructions, whereas previous SSE extensions work on 128-bit XMM registers. In addition to SIMD operations extending SSE's capabilities from 128- to 256-bit width, AVX brings to implementers:

- Non-destructive operations with 3- and 4-operand syntax (including for legacy 128-bit SIMD extensions);
- Relaxed memory alignment constraints, compared to SSE.

AVX operates on 256-bit SIMD registers called YMM, divided in two 128-bit lanes. The low lanes, i.e., the lower 128 bits, are aliased to the respective 128-bit XMM registers. Most instructions work "in-lane": each source element is applied only to other elements of the same lane. Some more expensive "cross-lane" instructions do exist, most notably shuffles.

AVX2 is an extension of AVX announced in 2011 that promotes most of the 128-bit SIMD integer instructions to 256-bit capabilities. AVX2 supports 4-way 64-bit integer addition, XOR, and vector shifts, thus enabling SIMD implementations of BLAKE-512. AVX2 also includes instructions to perform any-to-any permutation of words over a 256-bit register and vectorized table lookup to load elements in memory to YMM registers (see the instructions vperm* and vpgatherd* in §§4.2).

AVX is supported by Intel processors based on the Sandy Bridge microarchitecture (and future ones). The first processors commercialized were Core i7 and Core i5 in January 2011, but AVX will be (or is already) supported in new generations of Pentium and Celeron. AVX2 will be introduced in Intel's Haswell 22 nm architecture, to be released in 2013.

[7] See the benchmarks results on http://bench.cr.yp.to/results-sha3.html.

4.2 Useful AVX2 instructions

We focus on a small subset of the AVX2 instructions, presenting for each assembly instruction a brief explanation of what it does. For a better understanding, we also provide an equivalent description in C syntax using only general-purpose registers. Table 2 gives the respective C intrinsic functions.

Table 2. Intrinsics of main AVX2 instructions useful to implement BLAKE.

Assembly	Intrinsic	Description
`vpaddd`	`_mm256_add_epi32`	8-way 32-bit integer addition
`vpaddq`	`_mm256_add_epi64`	4-way 64-bit integer addition
`vpxor`	`_mm256_xor_si256`	XOR of the two 256-bit values
`vpsllvd`	`_mm256_sllv_epi32`	8-way 32-bit left-shift
`vpsllvq`	`_mm256_sllv_epi64`	4-way 64-bit left-shift
`vpsrlvd`	`_mm256_srlv_epi32`	8-way 32-bit right-shift
`vpsrlvq`	`_mm256_srlv_epi64`	4-way 64-bit right-shift
`vpermd`	`_mm256_permute8x32_epi32`	Shuffle of the eight 32-bit words
`vpermq`	`_mm256_permute4x64_epi64`	Shuffle of the four 64-bit words
`vpgatherdd`	`_mm256_i32gather_epi32`	8-way 32-bit table lookup
`vpgatherdq`	`_mm256_i32gather_epi64`	4-way 64-bit table lookup

`vpaddd` performs 8-way 32-bit integer addition, equivalently to

```
uint32_t a[8],b[8],c[8];
for(i=0; i < 8; ++i) c[i] = a[i] + b[i];
```

`vpaddq` performs 4-way 64-bit integer addition, equivalently to

```
uint64_t a[4],b[4],c[4];
for(i=0; i < 4; ++i) c[i] = a[i] + b[i];
```

`vpxor` performs a bitwise XOR over all words, representable in C as

```
uint32_t a[8],b[8],c[8];
for(i=0; i < 8; ++i) c[i] = a[i] ^ b[i];
```

`vpsllvd` performs 8-way 32-bit variable left-shift (that is, each of the eight 32-bit words can be rotated by a distinct value), equivalently to

```
uint32_t a[8],b[8],c[8];
for(i=0; i < 8; ++i) b[i] = a[i] << c[i];
```

Similarly, `vpsrlvd` performs 8-way 32-bit variable right-shift, and `vpsllvq` and `vpsrlvq` perform 4-way 64-bit variable left- and right-shift.

`vpermd` shuffles 32-bit words of a full YMM register across lanes using two YMM registers as inputs: one as source, the other as the permutation's indices:

```
uint32_t a[8],b[8],c[8];
for(i=0; i < 8; ++i) c[i] = a[b[i]];
```

`vpermq` is similar to `vpermd` but shuffles 64-bit words and takes an immediate operand instead as the permutation:

```
uint64_t a[4],c[4]; int b;
for(i=0; i < 4; ++i) c[i] = a[(b>>(2*i))%4];
```

`vpgatherdd` is among the most remarkable of the AVX2 extensions: it performs eight table lookups in parallel, as in the code below:

```
uint8_t *b; uint32_t scale, idx[8], c[8];
for(i=0; i < 8; ++i) c[i] = *(uint32_t)(b + idx[i]*scale);
```

vpgatherdq is quite similar to vpgatherdd, but works on four 64-bit words:

```
uint8_t *b; uint32_t scale, idx[4]; uint64_t c[4];
for(i=0; i < 4; ++i) c[i] = *(uint32_t)(b + idx[i]*scale);
```

4.3 Performance estimation

As previously mentioned, processors carrying the AVX2 instruction set are only expected to be available in 2013. There is currently no hard data on the performance of the instructions described above; one can, however, make some educated guesses, by using the Sandy Bridge as starting point.

The vpaddd, vpaddq, vpsllvd, vpsllvq, vpsrlvd, vpsrlvq, and vpxor instructions' performance can be expected to be on-par with Sandy Bridge's vxorps instruction, which requires a single cycle to complete. The vpermd and vpermq instructions cross register lanes; on Sandy Bridge, this adds one extra cycle of latency. We can estimate that this penalty gets no worse on Haswell, and that vpermd and vpermq require two cycles to complete. The gather instructions remain the most elusive; it is unknown whether this instruction consists of a large number of micro-ops, or uses dedicated circuitry. Assuming only one cache-line is accessed, one can expect at least three cycles of latency for the memory load, plus two for the extra logic.

We speculate that instruction parallelism in AVX2-compatible processors will resemble existing SSE2 parallelism available in current processors. Current Sandy Bridge processors are capable of executing three AVX instructions per cycle, namely one floating-point multiply, one floating-point add, and one logical operation. We expect future processors to be able to sustain such throughput with integer instructions, as it happens today with XMM registers.

5 Implementing BLAKE-512 with AVX2

We present a simple SIMD implementation of BLAKE-512, using AVX2's 4-way 64-bit SIMD instructions exactly in the same way that BLAKE-256 uses SSE2's 4-way 32-bit instructions. We then present some potential speed optimizations exploiting instructions proper to AVX2. The resulting performance can only be roughly estimated, for AVX2 is not yet fully documented (see §§4.3).

5.1 Straightforward SIMD implementation

AVX2 provides instructions to write a straightforward SIMD implementation of BLAKE-512 similar to the sse2 implementation of BLAKE-256 in §3, except that 256-bit YMM registers are used to hold four 64-bit words instead of 128-bit XMM registers being used to hold four 32-bit words.

The code below implements the column step of BLAKE-512's round function, i.e. it computes the first four instance of G in parallel. The 4×4 state of 64-bit words is stored in four YMM registers defined as __m256i type and aliased row1, row2, row3, and row4.

```
buf1 = _mm256_set_epi64x(m[sig[r][6]], m[sig[r][4]],
                         m[sig[r][2]], m[sig[r][0]]);
buf2 = _mm256_set_epi64x(u[sig[r][7]], u[sig[r][5]],
                         u[sig[r][3]], u[sig[r][1]]);
buf1 = _mm256_xor_si256(buf1, buf2);
row1 = _mm256_add_epi64(_mm256_add_epi64( row1, buf1), row2);
row4 = _mm256_xor_si256(row4, row1);
row4 = _mm256_xor_si256(_mm256_srli_epi64(row4, 32),
                        _mm256_slli_epi64(row4, 32));
row3 = _mm256_add_epi64(row3, row4);
row2 = _mm256_xor_si256(row2, row3);
```

```
buf1 = _mm256_set_epi64x(u[sig[r][6]], u[sig[r][4]],
                         u[sig[r][2]], u[sig[r][0]]);
buf2 = _mm256_set_epi64x(m[sig[r][7]], m[sig[r][5]],
                         m[sig[r][3]], m[sig[r][1]]);
buf1 = _mm256_xor_si256(buf1, buf2);
row2 = _mm256_xor_si256(_mm256_srli_epi64(row2, 25),
                        _mm256_slli_epi64(row2, 39));
row1 = _mm256_add_epi64(_mm256_add_epi64(row1, buf1), row2 );
row4 = _mm256_xor_si256(row4, row1);
row4 = _mm256_xor_si256(_mm256_srli_epi64(row4, 16),
                        _mm256_slli_epi64(row4, 48));
row3 = _mm256_add_epi64(row3, row4);
row2 = _mm256_xor_si256(row2, row3);
row2 = _mm256_xor_si256(_mm256_srli_epi64(row2, 11),
                        _mm256_slli_epi64(row2, 53));
row2 = _mm256_permute4x64_epi64(row2, _MM_SHUFFLE(0,3,2,1));
row3 = _mm256_permute4x64_epi64(row3, _MM_SHUFFLE(1,0,3,2));
row4 = _mm256_permute4x64_epi64(row4, _MM_SHUFFLE(2,1,0,3));
```

Excerpts of our assembly implementation can be found in Appendix C.1.

5.2 Rotations with shuffles

A simple optimization consists in implementing the rotation by 32 bits using the `vpshufd` instruction, which implements "in-lane" shuffle of 32-bit words. That is, the line

```
row4 = _mm256_xor_si256(_mm256_srli_epi64(row4, 32),
                        _mm256_slli_epi64(row4, 32));
```

is replaced by

```
row4 = _mm256_shuffle_epi32(row4, _MM_SHUFFLE(2,3,0,1));
```

Similarly, the rotations by 16 bits can be implemented using `vpshufb` in a similar fashion as §§3.3:

```
row4 = _mm256_shuffle_epi8(row4, r16);
```

where `r16` is the alias of a YMM register containing the index values for the byte of `row4` at its respective lane and position.

Based on the estimates in §§4.3, we expect to save two cycles per rotation of 16 or 32 bits, i.e. four cycles per column step, eight cycles per round, or 128 cycles per compression function.

5.3 Message loads with `vpgatherdq`

As observed in §§4.2, the `vpgatherdq` instruction can be used to load words from arbitrary memory addresses. To load message words according to the σ_r permutation, one would thus write the following code:

```
_m256i m0 = _mm_i32gather_epi64(m, sigma[r][0], 8);
_m256i m1 = _mm_i32gather_epi64(m, sigma[r][1], 8);
_m256i m2 = _mm_i32gather_epi64(m, sigma[r][2], 8);
_m256i m3 = _mm_i32gather_epi64(m, sigma[r][3], 8);
```

where `sigma[r][i]`'s are __m128i type, and where each 32-bit word holds an index of the permutation. As each `sigma[r][i]` holds four indices, `sigma[r][0]` to `sigma[r][3]` hold the 16 indices of the σ_r permutation.

Such a sequential implementation of four `vpgatherdq`'s is expected to only add an extra latency equivalent to that of a single `vpgatherdq`, for the subsequent instructions (XOR, etc.) only depend on the first call, and therefore will not stall while the three others are executed.

5.4 Message caching

As discussed in the previous section, loading the message words according to the σ permutation takes a considerable number of cycles, compared to an arithmetic operation. A potential optimization allows to reduce the cost of message loading, by observing that in BLAKE-512, six of the ten σ permutations are used twice. That is, a same permuted message is used twice for the permutations $\sigma_0, \sigma_1, \dots, \sigma_5$. An implementation strategy could thus be:

1. In rounds $0, \dots, 5$: compute the permuted messages, and store the result (preferably in unused YMM registers);
2. In rounds $6, \dots, 9$: compute the permuted messages without storing the result;
3. In rounds $10, \dots, 15$: do not compute the permuted messages, but rather use the registers set in step 1.

To save an additional XOR, one should store the permuted message already XORed with the constants.

Unfortunately, the above strategy would require 24 YMM registers only to store the permuted message—as a BLAKE-512 message block is 1024-bit, occupying four YMM registers—whereas only 16 are available and at least six are necessary to implement the round function.

Nevertheless, 24 YMM registers represent 768 bytes of memory, which fits comfortably in most processors' L1 cache. The ≈ 3-cycle penalty for L1 accesses should be easily avoidable by loading the messages early. In anything other than synthetic benchmarks, it is possible that the resulting cache evictions can diminish the overall performance.

Note that message caching is not proper to AVX2, but can also be used by SSE implementations of BLAKE-512.

5.5 Performance estimates

Based on estimations in §§4.3, one can attempt to predict the speed of an implementation of BLAKE-512 using the aforementioned techniques. For simplicity, we assume that no message caching is used. We first propose a pessimistic estimate—i.e. likely to overestimate the cycles/byte count—based on the following assumptions:

- Message loading takes 5 cycles per step (latency of one `vpgatherdq`).
- Each addition or XOR consumes one cycle, and all are computed serially.
- XORs between message and constants are computed in parallel to other operations (message loads or shuffles), and thus are not counted.
- Rotations by 32 or 16 bits take one cycle, while those by 25 or 11 bits take three cycles.
- Of the three shuffles performed for (un)diagonalization, two are computed in parallel to additions or XORs.
- Initialization and finalization of the compression function add an overhead of respectively 10 and 6 cycles.

This estimates each round to $2 \times (5 + 10 + 2 + 6 + 2) = 52$ cycles, thus $52 \times 16 + 16 = 848$ cycles for the compression function, i.e. 6.63 cycles per byte of message.

The above assumes that from the second round, message loading waits for the end of the previous round to start. We may thus (optimistically) assume that `vpgatherdq` will run in parallel to the end of the previous round, possibly leading to a higher speed.

We stress that that only actual benchmarks on real hardware would provide reliable speed figures. We only make the above estimates as an attempt to predict the real speed based on the data available and on our experiments with AVX.

6 Implementing BLAKE-256

This section shows how BLAKE-256 can benefit of AVX2. Unlike BLAKE-512, BLAKE-256 is not naturally adaptable to 256-bit vectors, as there is a maximum of four G_i independently-running functions per round. Nevertheless, it is possible to take advantage of AVX2 to speedup BLAKE-256. Excerpts of our assembly implementation appear in Appendix C.3.

6.1 Optimizing message loads with AVX2

The first way to improve message loads is by using the vpgatherdd instruction from the AVX2 instruction set. To perform the full 16-word message permutation required in each round, only four operations are required:

```
_m128i m0 = _mm_i32gather_epi32(m, sigma[r][0], 4);
_m128i m1 = _mm_i32gather_epi32(m, sigma[r][1], 4);
_m128i m2 = _mm_i32gather_epi32(m, sigma[r][2], 4);
_m128i m3 = _mm_i32gather_epi32(m, sigma[r][3], 4);
```

This can be further improved by using only two YMM registers to store the permuted message:

```
_m256i m01 = _mm256_i32gather_epi32(m, sigma[r][0], 4);
_m256i m23 = _mm256_i32gather_epi32(m, sigma[r][1], 4);
```

The individual 128-bit blocks of message are then accessible through the vextracti128 instruction, e.g.:

```
m1 = _mm256_extracti128_si256(m01, 1);
```

One must also consider the possibility that vpgatherdd will not have acceptable performance, perhaps due to specific processor design idiosyncrasies; AVX2 can still help us, via the vpermd and vpblendd instructions:

```
tmp0 = _mm256_permutevar8x32_epi32(m01, sigma00);
tmp1 = _mm256_permutevar8x32_epi32(m23, sigma01);
tmp2 = _mm256_permutevar8x32_epi32(m01, sigma10);
tmp3 = _mm256_permutevar8x32_epi32(m23, sigma11);
m01  = _mm256_blend_epi32(tmp0, tmp1, mask0);
m23  = _mm256_blend_epi32(tmp2, tmp3, mask1);
```

In the above code, we permute the elements from the first YMM register into their proper order in the permutation, after which we permute the elements from the second. A simple blend instruction suffices to obtain the correct permutation. We repeat the process for the second part of the permutation. Once again, individual 128-bit blocks are available via vextracti128.

For completeness, the instructions vextracti128 and vpblendd are documented in Appendix B.

6.2 Optimizing message loads with XOP

The XOP instruction set is AMD's extension to AVX (see Appendix A). An approach similar to that in §§6.1 can be used with the XOP instruction set, via the vpperm and vpinsrd instructions: We note that no 4-word subgroup of the ten permutations of BLAKE-256 requires words from all four 128-bit blocks of the message; the most common occurrence is that two words come from one block, and one word out of two other 128-bit blocks. For this case, we can use the following approach:

```
m0 = _mm_perm_epi8(m0, m1, sel0);
m0 = _mm_insert_epi32(m0, m.u32[4], 3);
m1 = _mm_perm_epi8(m3, m1, sel1);
m1 = _mm_insert_epi32(m1, m.u32[5], 6);
m2 = _mm_perm_epi8(m0, m1, sel2);
m2 = _mm_insert_epi32(m2, m.u32[7], 1);
m3 = _mm_perm_epi8(m0, m1, sel3);
m3 = _mm_insert_epi32(m3, m.u32[14], 9);
```

The few cases where only two 128-bit blocks need to be accessed can be dealt with with a single vpperm instruction.

The vpinsrd instruction is documented in Appendix B.

6.3 Message caching

Like BLAKE-512, BLAKE-256 reuses several permuted messages, namely four. Due to the smaller number of redundant permuted messages and the smaller messages, this full state ($4 \times 4 \times 128$) can be stored in eight YMM registers. This leaves the possibility of either storing all entries, or to keep some in registers. Permuted messages are easily stored using the `vinserti128` instruction:

```
// First 4 permuted elements
cache_reg = _mm256_inserti128_si256(cache_reg, buf1, 0);
...
// Second 4 permuted elements
cache_reg = _mm256_inserti128_si256(cache_reg, buf1, 1);
_mm256_store_si256(&cache[r], cache_reg);
```

In rounds 10 and above, we can retrieve the cached permutations with a simple load and extract:

```
cache_reg = _mm256_load_si256(&cache[r]);
buf1 = _mm_extracti128(cache_reg, 0);
...
buf1 = _mm_extracti128(cache_reg, 1);
```

Like for BLAKE-512, one should store the message words already XORed with the constants, to save the extra XORs.

6.4 Performance estimates

As BLAKE-256 seems to marginally benefit from AVX2, we focused our performance estimation on the AVX assembly implementation—the AVX2 is unlikely to be slower, as AVX is a subset of AVX2. Excerpts from our AVX implementation are given in Appendix C.2.

To measure the speed of our AVX implementation on long messages, we reused tools from SUPERCOP [1], and produced our executable as follows:

```
yasm -f elf64 b256.asm
$(CC) $(FLAGS) $(INCLUDE) -o measure hash.c b256.o measure-anything.c amd64cpuinfo.c \
measure.c -DCOMPILER=\"$(CC)\" -DLOOPS=3 -DSUPERCOP
```

where `$(CC)=gcc`, `$(FLAGS)=-static -m64 -march=native -mtune=native -O2 -fomit-frame-pointer`, and `$(INCLUDE)` are SUPERCOP-related includes.

We ran benchmarks on an Intel Core i7 2630QM (2 GHz, Sandy Bridge architecture), and measured a speed of 7.62 cpb. For comparison, at the time of writing the latest benchmarks from eBASH on `sandy0`[8] (`supercop-20110708`) report BLAKE-256 running at 7.87 cpb (with the `sse41` C implementation).

Surprisingly, message caching did not bring a significant speed-up. We will submit our code to SUPERCOP for independent benchmarks on other machines.

References

1. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to/` Accessed 24 August 2011.
2. Intel: Advanced vector extensions programming reference (June 2011) Document no. 319433-011.
3. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. Submission to the SHA-3 competition (2008) `http://www.131002.net/blake/`.
4. Intel: C++ Intrinsics Reference (2007) Document no. 312482-002US.
5. Coke, J., Baliga, H., Cooray, N., Gamsaragan, E., Smith, P., Yoon, K., Abel, J., Valles, A.: Improvements in the Intel Core 2 Penryn Processor Family Architecture and Microarchitecture. Intel Technology Journal **12**(3) (October 2008) 179–193
6. Intel: Advanced vector extensions programming reference (March 2008) Document no. 319433-002.
7. AMD: AMD64 Architecture Programmers Manual Volume 6: 128-Bit and 256-Bit XOP, FMA4 and CVT16 Instructions (November 2009)

[8] CPU details: Sandy Bridge (206a7); 2011 Intel Core i7-2600K; 4 x 3400 MHz.

A The XOP instruction set

In 2007, AMD announced its SSE5 set of new instructions. These featured 3-operand instructions, more powerful permutations, native integer rotations, and fused-multiply-add capabilities. After the announcement of AVX, however, SSE5 was shelved in favor of AVX plus XOP, FMA4, and CVT16.

The XOP instruction set [7] extends AVX with new integer multiply-and-accumulate (`vpmac*`), rotation (`vprot*`), shift (`vpsha*`, `vpshl*`), permutation (`vpperm`), and conditional move (`vpcmov`) instructions working on XMM registers. The rotations, in particular, are of obvious advantage to BLAKE. The first chip implementing XOP is AMD's FX-8150 Bulldozer (32 nm), first released to the market on October 12, 2011.

B Reference of additional instructions

This section briefly documents the AVX2 instructions `vextracti128`, `vpblendd`, and `vpinsrd` used in §§6.1 and §§6.2.

The `vpinsrd` instruction, also accessible by its intrinsic `_mm_insert_epi32`, inserts one 32-bit word into a specified position in a XMM register, as the following code illustrates:

```
uint32_t c[8], a; int imm;
c[imm] = a;
```

`vpblendd`, similar to the SSE4.1 `pblendw` instruction, permits the selection of words from 2 different sources according to an immediate index, placing them in a third destination register:

```
uint32_t a[8], b[8], c[8]; int sel;
for(i=0; i < 8; ++i)
    if((sel>>i)&1) c[i] = b[i];
    else c[i] = a[i];
```

`vextracti128` (`_mm256_extracti128_si256`) and `vinserti128` (`_mm256_inserti128_si256`) extract and insert an XMM register into the lower or upper halves of a YMM register. `vextracti128` is equivalent to:

```
uint32_t a[8], c[4]; int imm;
for(i=0; i < 4; ++i) c[i] = a[i + 4*imm];
```

while `vinserti128` is equivalent to

```
uint32_t a[8], b[4], c[8]; int imm;
for(i=0; i < 8; ++i) c[i] = a[i];
for(i=0; i < 4; ++i) c[i+4*imm] = b[i];
```

C Assembly implementations (excerpts)

This section presents excerpts from our assembly implementations. The full implementations will be made publicly available.

C.1 Assembly implementation of BLAKE-512 for AVX2

Implementation of G, with permuted message in %3 and %4:

```
; Helper word rotation macro
%macro VPROTRQ 2
    vpsllq  ymm8, %1, 64-%2 ; x << 32-c
    vpsrlq  %1, %1, %2  ; x >> c
    vpxor   %1, %1, ymm8
%endmacro
```

```
; ymm0-3: State
; ymm4-7: m_{\sigma} xor c_{\sigma}
; ymm8-9: Free temp registers
; ymm10-13: m
%macro G 2
        vpaddq  ymm0, ymm0, %1 ; row1 + buf1
        vpaddq  ymm0, ymm0, ymm1 ; row1 + row2
        vpxor   ymm3, ymm3, ymm0 ; row4 ^ row1
        vpshufd ymm3, ymm3, 10110001b ; row4 >>> 32

        vpaddq  ymm2, ymm2, ymm3 ; row3 + row4
        vpxor   ymm1, ymm1, ymm2 ; row2 ^ row3
        VPROTRQ ymm1, 25   ; row2 >>> 25

        vpaddq  ymm0, ymm0, %2 ; row1 + buf1
        vpaddq  ymm0, ymm0, ymm1 ; row1 + row2
        vpxor   ymm3, ymm3, ymm0 ; row4 ^ row1
        vpshufb ymm3, ymm3, ymm15 ; row4 >>> 16

        vpaddq  ymm2, ymm2, ymm3 ; row3 + row4
        vpxor   ymm1, ymm1, ymm2 ; row2 + row3
        VPROTRQ ymm1, 11    ; row2 >>> 11
%endmacro
```

Message loading:

```
%macro MSGLOAD 1

    vpcmpeqq   ymm14, ymm14, ymm14 ; FF..FF
    vmovdqa    xmm8, [perm + %1*64 + 00]
    vpgatherdq ymm4, [rsp + 8*xmm8], ymm14

    vpcmpeqq   ymm14, ymm14, ymm14 ; FF..FF
    vmovdqa    xmm9, [perm + %1*64 + 16]
    vpgatherdq ymm5, [rsp + 8*xmm9], ymm14

    vpcmpeqq   ymm14, ymm14, ymm14 ; FF..FF
    vmovdqa    xmm8, [perm + %1*64 + 32]
    vpgatherdq ymm6, [rsp + 8*xmm8], ymm14

    vpcmpeqq   ymm14, ymm14, ymm14 ; FF..FF
    vmovdqa    xmm9, [perm + %1*64 + 48]
    vpgatherdq ymm7, [rsp + 8*xmm9], ymm14

    vpxor      ymm4, ymm4, [const_z + 128*%1 + 00]
    vpxor      ymm5, ymm5, [const_z + 128*%1 + 32]
    vpxor      ymm6, ymm6, [const_z + 128*%1 + 64]
    vpxor      ymm7, ymm7, [const_z + 128*%1 + 96]

%ifdef CACHING
%if %1 < 6
    vmovdqa    [rsp + 128 + %1*128 + 00], ymm4
    vmovdqa    [rsp + 128 + %1*128 + 32], ymm5
    vmovdqa    [rsp + 128 + %1*128 + 64], ymm6
    vmovdqa    [rsp + 128 + %1*128 + 96], ymm7
%endif
%endif

%endmacro
```

Diagonalization, undiagonalization, and a round:

```
%macro DIAG 0
        vpermq  ymm1, ymm1, 0x39
        vpermq  ymm2, ymm2, 0x4e
        vpermq  ymm3, ymm3, 0x93
%endmacro

%macro UNDIAG 0
        vpermq  ymm1, ymm1, 0x93
        vpermq  ymm2, ymm2, 0x4e
        vpermq  ymm3, ymm3, 0x39
%endmacro

%macro ROUND 1
    MSGLOAD %1
    G ymm4, ymm5
    DIAG
    G ymm6, ymm7
    UNDIAG
%endmacro
```

C.2 Assembly implementation of BLAKE-256 for AVX

Our AVX code implements the message loading for each permutation separately, with option support for message caching. Below we show the code for the second permutation:

```
%macro MSGLOAD1 0
;m[ 3] m[ 2]  m[ 1]  m[ 0] -> m[13] m[ 9] m[ 4] m[14]
;m[ 7] m[ 6]  m[ 5]  m[ 4] -> m[ 6] m[15] m[ 8] m[10]
;m[11] m[10]  m[ 9]  m[ 8] -> m[ 5] m[11] m[ 0] m[ 1]
;m[15] m[14]  m[13]  m[12] -> m[ 3] m[ 7] m[ 2] m[12]
; xmm7 xmm6 xmm5 xmm4 <- xmm13 xmm12 xmm11 xmm10

vpalignr xmm4, xmm11, xmm13, 4 ; 7 6 5 | 4 15 14 13 | 12
vpinsrd  xmm4, xmm4, [rsp + 9*4], 2 ; 4 9 14 13
vpshufd  xmm4, xmm4, 00101101b ; 13 9 4 14

vpblendw xmm5, xmm12, xmm13, 11000000b ; 15 10 9 8
vpinsrd  xmm5, xmm5, [rsp + 6*4], 1 ; 15 10 6 8
vpshufd  xmm5, xmm5, 01110010b ; 6 15 8 10

vpunpcklqdq xmm6, xmm11, xmm10 ; 1 0 5 4
vpinsrd  xmm6, xmm6, [rsp + 11*4], 0 ; 1 0 5 11
vpshufd  xmm6, xmm6, 01001011b ; 5 11 0 1

vpunpckhdq xmm7, xmm11, xmm10 ; 3 7 2 6
vpblendw xmm7, xmm7, xmm13, 00000011b ; 3 7 2 12

vpxor xmm4, xmm4, [const_z + 1*64 + 00]
vpxor xmm5, xmm5, [const_z + 1*64 + 16]
vpxor xmm6, xmm6, [const_z + 1*64 + 32]
vpxor xmm7, xmm7, [const_z + 1*64 + 48]

%ifdef CACHING
vmovdqa [rsp + 16*4 + 1*64 + 00], xmm4
vmovdqa [rsp + 16*4 + 1*64 + 16], xmm5
vmovdqa [rsp + 16*4 + 1*64 + 32], xmm6
vmovdqa [rsp + 16*4 + 1*64 + 48], xmm7
```

```
%endif

%endmacro
```

The first ten rounds are implemented as:

```
%macro ROUND 1
        MSGLOAD%1
        G xmm4, xmm5
        DIAG
        G xmm6, xmm7
        UNDIAG
%endmacro
```

When message caching is activated, the last four rounds are directly loading the message words xored with constants from memory:

```
%macro ROUNDC 1
        G [rsp + 16*4 + 64*%1 + 00], [rsp + 16*4 + 64*%1 + 16]
        DIAG
        G [rsp + 16*4 + 64*%1 + 32], [rsp + 16*4 + 64*%1 + 48]
        UNDIAG
%endmacro
```

C.3 Assembly implementation of BLAKE-256 for AVX2

AVX2 allows the use of `vpgatherdd` for direct load of permuted message words from memory:

```
%macro MSGLOAD 1

vpcmpeqd ymm12, ymm12, ymm12
vmovdqa    ymm8, [perm + %1*64 + 00]
vpgatherdd ymm4, [ymm8*4+rsp], ymm12

vpcmpeqd ymm13, ymm13, ymm13
vmovdqa    ymm9, [perm + %1*64 + 32]
vpgatherdd ymm6, [ymm9*4+rsp], ymm13

vpxor ymm4, ymm4, [const_z + %1*64 + 00]
vpxor ymm6, ymm6, [const_z + %1*64 + 32]

%ifdef CACHING
%if %1 < 4
vmovdqa [rsp + 16*4 + %1*64 + 00], ymm4
vmovdqa [rsp + 16*4 + %1*64 + 32], ymm6
%endif
%endif

; Unpack into XMM
vextracti128 xmm5, ymm4, 1
vextracti128 xmm7, ymm6, 1

%endmacro
```