Achieving high performance for Advanced Encryption Standard (AES) applications

How to use the IBM POWER8 built-in functions of IBM XL C/C++ and Fortran compilers

Anh Tuyen Tran

April 02, 2015

This article provides a detailed analysis about the new Advanced Encryption Standard (AES) built-in functions provided by IBM® XL compilers. It explores the code that the compilers generate to demonstrate the performance gained by using the built-in functions. This is critical for high-performance AES applications, especially when many of them cannot be parallelized due to the nature of operation modes, such as chaining mode [for example, cipher block chaining (CBC) and propagating cipher block chaining (PCBC)] or feedback mode [cipher feedback (CFB) and output feedback (OFB)].

If you are a software engineer who is interested in achieving high performance for applications on IBM POWER8™ processors, this article is for you.

Introduction

AES is a US government standard for encryption, and is widely used by the US government and industries around the world. It protects classified information and sensitive data. The new IBM XL compilers for C/C++ and Fortran provide support for AES built-in functions on both little endian and big endian systems. Using the IBM POWER8 built-in functions, AES applications can achieve high performance on IBM POWER8 processors.

Generally available in 2014, IBM XL compilers deliver support for POWER8 functionality, unleashing the power of the latest POWER8 processors. Among the new features that directly invoke the architecture instructions at the application level are the POWER8 cryptography built-in functions for AES. These built-in functions map each of the major AES operations, one-to-one, to the corresponding instruction at the hardware level on POWER8 processors. This article explores the new built-in cryptography functions for AES encryption and decryption supported by IBM XL compilers.

Versions of IBM XL compilers that support AES built-in functions

AES built-in functions are supported in the following IBM XL compilers:
• C/C++ version 13 and higher
• Fortran version 15 and higher

Depending on the platform, the versions of the compilers supporting POWER8 features and enhancements are shown in Table 1.

Table 1. Versions of XL compilers with POWER8 built-in functions

<table>
<thead>
<tr>
<th>Language</th>
<th>Big endian</th>
<th>Little endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>XL C/C++ V13.1</td>
<td>XL C/C++ V13.1.1</td>
</tr>
<tr>
<td></td>
<td>OS: AIX®, Linux®</td>
<td>OS: Linux</td>
</tr>
<tr>
<td>Fortran</td>
<td>XL Fortran V15.1</td>
<td>XL Fortran V15.1.1</td>
</tr>
<tr>
<td></td>
<td>OS: AIX, Linux</td>
<td>OS: Linux</td>
</tr>
</tbody>
</table>

A primer of AES

In May 2002, AES became the US government standard for encryption. It was the first publicly accessible and open cipher approved by the US National Security Agency (NSA) to protect classified information up to the top secret level when used in an NSA approved cryptographic module. AES is a block cipher algorithm that uses a combination of both substitution and permutation on blocks of text and a key in order to produce the output blocks. Each block, also called a state, is a 4x4 column-major matrix of bytes (128-bit value).

Note: The column-major matrix of a state is stipulated by AES and is independent of the language implementation of native storage orders in the memory.

Now, let's take a look at how AES works and how the architecture maps to specific POWER8 built-in functions.

An example of the state representation of an input text stream is shown in Figure 1.

Figure 1. State representation of a text stream

AES is a symmetric key algorithm that uses the same key for both encryption and decryption processes. The legal values for the length of an AES key are 16, 24, and 32 in bytes or equivalently 128, 192 and 256 in bits. These values correspond to the key length (Nk) values 4, 6 and 8, specified by the standard. The key length used by an AES application dictates the number of transformation rounds to convert plain text to cipher text, and from cipher text to plain text. The number of rounds (Nr) computation is as follows:

Number of rounds (Nr) = 10 + (Key length in -bytes – 16) / 4

The key length, block size, and the number of rounds for 128-bit, 192-bit, and 256-bit AES are displayed in Table 2.
Table 2. Legal values for 128-bit, 192-bit, and 256-bit AES

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Key length (in bits)</th>
<th>Key length (in bytes)</th>
<th>Key length (Nk words)</th>
<th>Block size (Nb words)</th>
<th>Number of rounds (Nr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-bit AES</td>
<td>128</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>192-bit AES</td>
<td>192</td>
<td>24</td>
<td>6</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>256-bit AES</td>
<td>256</td>
<td>32</td>
<td>8</td>
<td>4</td>
<td>14</td>
</tr>
</tbody>
</table>

The key for each round is derived from the given cipher key using the Rijndael [an algorithm selected by the US National Institute of Standards and Technology (NIST) as for AES] key schedule. As shown in Figure 2, regardless of the length of the key in use, each round key is a 4x4 column-major matrix of bytes, which is 128 bits in length. Note that IBM XL compilers do not provide a built-in function for key expansion.

Figure 2. Relationship between key expansion and rounds in AES encryption

The initial round of AES encryption consists of only AddRoundKey() transformation, where a round key is added to the state by an XOR operation. All other rounds, with the exception of the last round, consist of four transformations: SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey(). The final round is slightly different: it only has SubBytes(), ShiftRows() and AddRoundKey().
Although using the same cipher key as AES encryption, AES decryption is slightly different. It has the initial round consisting of only \texttt{AddRoundKey()} transformation. All intermediate rounds consist of \texttt{InverseShiftRows()}, \texttt{InverseSubBytes()}, \texttt{AddRoundKey()}, and \texttt{InverseMixColumns()}. The final round only has \texttt{InverseShiftRows()}, \texttt{InverseSubBytes()}, and \texttt{AddRoundKey()}.

The symmetry between the corresponding operations in AES encryption and AES decryption is displayed in Figure 3.

\textbf{Figure 3. Symmetry between AES encryption and AES decryption}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{Figure3.png}
\end{figure}

\textbf{IBM XL compiler built-in functions for AES}

IBM POWER8 architectures provide hardware level support for the most critical functions used by AES. IBM XL compilers provide a set of cryptography built-in functions, mapping one-to-one to the hardware instructions. They are alternatives to managing hardware registers through assembly language. They provide access to these optimized instruction sets for software engineers. The IBM XL compiler built-in functions for AES include:

- \texttt{vcipher(state_array, round_key)}
- \texttt{vcipherlast(state_array, round_key)}
- \texttt{vncipher(state_array, round_key)}
- \texttt{vncipherlast(state_array, round_key)}
- \texttt{vsbox(state_array)}

\texttt{vcipher(state_array, round_key)}

This built-in function performs one round of the AES cipher operation on intermediate \texttt{state_array} using a given \texttt{round_key}. This is used for all rounds during the encryption process, except the
initial and the final rounds. The interfaces of the vcipher built-in function in C/C++, Fortran and assembly languages are listed in this section.

- **C/C++**
  vector unsigned char __vcipher (vector unsigned char state_array, vector unsigned char round_key)

- **Fortran**
  VCIIPHER (STATE_ARRAY, ROUND_KEY) where both arguments and the result are unsigned vectors of kind 1.

- **Assembly**
  vcipher VRT,VRA,VRB
  - VRT holds the result, representing the new intermediate state of the cipher operation.
  - VRA holds the state, representing the intermediate state array during AES cipher operation.
  - VRB holds the round key.

**vcipherlast (state_array, round_key)**

This built-in function performs a final round of the AES cipher operation on intermediate state_array using a given round_key. This is used for the final round of the encryption process. Its output is the ciphertext. The interfaces of the vcipherlast built-in function in C/C++, Fortran and assembly languages are listed in this section.

- **C/C++**
  vector unsigned char __vcipherlast (vector unsigned char state_array, vector unsigned char round_key)

- **Fortran**
  VCIPHERLAST (STATE_ARRAY, ROUND_KEY) where both arguments and the result are unsigned vectors of kind 1.

- **Assembly**
  vcipherlast VRT,VRA,VRB
  - VRT holds the result, representing the final state of the cipher operation.
  - VRA holds the state, representing the intermediate state array during AES cipher operation.
  - VRB holds the round key.

The pseudo codes of AES encryption when using user-defined functions and using POWER8 built-in functions are shown in Figure 4.
Achieving high performance for Advanced Encryption Standard (AES) applications

Figure 4. Relationship between AES operations and AES built-in functions in AES encryption

<table>
<thead>
<tr>
<th>AES encryption as specified in Federal Information Processing Standards Publication 197 (FIPS-197)</th>
<th>AES encryption using XL built-in cryptography functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt (byte in[4<em>Nb], byte out[4</em>Nb], word w[Nb*(Nt+1)]) begin</td>
<td>Encrypt (byte in[4<em>Nb], byte out[4</em>Nb], word w[Nb*(Nt+1)]) begin</td>
</tr>
<tr>
<td>/* The initial round */</td>
<td>/* The initial round */</td>
</tr>
<tr>
<td>AddRoundKey(state, w[0, Nb-1]) // Sec. 5.1.4 of FIPS-197</td>
<td>AddRoundKey(state, w[0, Nb-1]) // Sec. 5.1.4 of FIPS-197</td>
</tr>
<tr>
<td>/* Rounds 1 to round Nt-1 */</td>
<td>/* Rounds 1 to round Nt-1 */</td>
</tr>
<tr>
<td>for round = 1 step 1 to Nt-1</td>
<td>for round = 1 step 1 to Nt-1</td>
</tr>
<tr>
<td>SubBytes(state) // Sec. 5.1.1 of FIPS-197</td>
<td>VCRYPTER(state, w(round+1)*Nb-1)</td>
</tr>
<tr>
<td>ShiftRows(state) // Sec. 5.1.2 of FIPS-197</td>
<td>/* Rounds 1 to round Nt-1 */</td>
</tr>
<tr>
<td>MixColumns(state) // Sec. 5.1.3 of FIPS-197</td>
<td>for round = 1 step 1 to Nt-1</td>
</tr>
<tr>
<td>AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])</td>
<td>VCRYPTER(tree, w(round+1)*Nb-1)</td>
</tr>
<tr>
<td>end for</td>
<td>end for</td>
</tr>
<tr>
<td>/* The final round */</td>
<td>/* The final round */</td>
</tr>
<tr>
<td>SubBytes(state)</td>
<td>VCRYPTERLAST(state, w[Nt]*Nb, (Nt+1)*Nb-1)</td>
</tr>
<tr>
<td>ShiftRows(state)</td>
<td></td>
</tr>
<tr>
<td>AddRoundKey(state, w[Nt]*Nb, (Nt+1)*Nb-1)</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td></td>
</tr>
</tbody>
</table>

vncipher (state_array, round_key):

This built-in function performs one round of the AES decipher operation on intermediate state_array using a given round_key. This is used for all rounds during the decryption process, except the initial and the final rounds. The interfaces of the vncipher built-in function in C/C++, Fortran and assembly languages are listed in this section.

- **C/C++**
  
  ```c
  vector unsigned char __vncipher (vector unsigned char state_array, vector unsigned char round_key)
  ```

- **Fortran**
  
  ```fortran
  VCRYPTER (STATE_ARRAY, ROUND_KEY) where both arguments and the result are unsigned vectors of kind 1.
  ```

- **Assembly**
  
  ```assembly
  vncipher VRT,VRA,VRB
  ```
  
  - VRT holds the result, representing the new intermediate state of the inverse cipher operation.
  - VRA holds the State, representing the intermediate state array during AES inverse cipher operation.
  - VRB holds the round key.

vncipherlast (state_array, round_key)
This function performs a final round of the AES decipher operation on intermediate \textit{state\_array} using a given \textit{round\_key}. This is used for the final round of the decryption process. Its output is the original plain text. The interfaces of the \texttt{vncipherlast} built-in function in C/C++, fortran and Assembly languages are listed in this section.

- **C/C++**
  
  \begin{verbatim}
  vector unsigned char __vncipherlast(vector unsigned char state_array, vector unsigned char round_key)
  \end{verbatim}

- **Fortran**
  
  \texttt{VNCIPHERLAST (STATE\_ARRAY, ROUND\_KEY)} where both arguments and the result are unsigned vectors of kind 1.

- **Assembly**
  
  \texttt{vncipherlast VRT,VRA,VRB}
  
  - \texttt{VRA} holds the state, representing the intermediate state array during AES inverse cipher operation.
  - \texttt{VRB} holds the round key.
  - \texttt{VRT} holds the result, representing the final state of the inverse cipher operation.

The pseudo codes of AES decryption when using user-defined functions and using POWER8 built-in functions are presented in Figure 5.

**Figure 5. Relationship between AES operations and AES built-in functions in AES decryption**

<table>
<thead>
<tr>
<th>AES decryption as specified in Federal Information Processing Standards Publication 197 (FIPS-197)</th>
<th>AES decryption using XL built-in cryptography functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrypt (byte inf[4<em>Nb], byte out[4</em>Nb], word w[Nb*(N+1)]) begin /* The initial round <em>/ AddRoundKey(state, w[Nb</em>Nb, (N+1)<em>Nb-1]) // Sec. 5.1.4 of FIPS-197 /</em> Rounds N-1 to round 1 <em>/ for round = N-1 step -1 to 1 InvShiftRows(state) // Sec. 5.3.1 of FIPS-197 InvSubByte(state) // Sec. 5.3.2 of FIPS-197 AddRoundKey(state, w[round</em>Nb, (round+1)<em>Nb-1]) InvMixColumns(state) // Sec. 5.3.3 of FIPS-197 end for /</em> The final round */ InvShiftRows(state) InvSubByte(state) AddRoundKey(state, w[0, Nb-1]) End</td>
<td>Decrypt (byte inf[4<em>Nb], byte out[4</em>Nb], word w[Nb*(N+1)]) begin /* The initial round <em>/ AddRoundKey(state, w[Nb</em>Nb, (N+1)<em>Nb-1]) // Sec. 5.1.4 of FIPS-197 /</em> Rounds 1 to round N-1 <em>/ for round = 1 step 1 to Nb-1 VNCIPHER(state, w[round</em>Nb, (round+1)<em>Nb-1]) end for /</em> The final round */ VNCIPHERLAST(state, w[0, Nb-1]) End</td>
</tr>
</tbody>
</table>

The relationship between POWER8 built-in functions to AES encryption and AES decryption is illustrated in Figure 6.
In addition, there is a built-in function to perform the SubBytes( ) operation, as defined in FIPS-197, on state_array.

**vsbox (state_array)**

This built-in function performs the SubBytes( ) operation, a nonlinear byte substitution operating independently on each byte of the state using a pre-defined substitution table (S-box). The interfaces of the vsbox built-in function in C/C++, Fortran, and assembly languages are listed in this section.

- **C/C++**
  
  ```c
  vector unsigned char __vsbox(vector unsigned char state_array)
  ```

- **Fortran**
  
  ```fortran
  VSBOX(STATE_ARRAY) where the argument and result are unsigned vectors of kind 1.
  ```

- **Assembly**
  
  ```assembly
  vsbox VRT, VRA
  ```
  - VRA holds the state, representing the intermediate state array during AES cipher operation.
  - VRT holds the result of applying the transform, SubBytes() on state, as defined in FIPS-197.
Figure 7. S-box lookup operation

For example, if the input is \{68\}, then the substitution value would be determined by the intersection of the sixth row and the eighth column of the S-box. This would result in having an output value of \{45\}.

Test programs using cryptography built-in functions of IBM XL C/C++ V13.1 compiler

The following snippets use the XL C/C++ V13.1 cryptography built-in functions on POWER8 architecture.

- An intermediate round of AES encryption with _vcipher
- A SubBytes() option, that is, S-box lookup with _vsbox

Example of __vcipher

The input is the intermediate block and the round key listed in Appendix B of Federal Information Processing Standards Publication 197(FIPS-197).
At the beginning of round 1, the input is

\{ 0x19, 0x3d, 0xE3, 0xBE, 0xA0, 0xF4, 0xE2, 0x2B, 0x9A, 0xC6, 0x8D, 0x2A, 0xE9, 0xF8, 0x48, 0x08 \}

The round key corresponding to the round is

\{ 0xA0, 0xFA, 0xFE, 0x17, 0x88, 0x54, 0x2C, 0xB1, 0x23, 0xA3, 0x39, 0x39, 0x2A, 0x6C, 0x76, 0x05 \}

After going through \texttt{SubBytes()}, \texttt{ShiftRows()}, \texttt{MixColumns()}, and \texttt{AddRoundKey()} in round 1, the output will be

\{ 0xA4, 0x9C, 0x7F, 0xF2, 0x68, 0x9F, 0x35, 0x2B, 0x6B, 0x5B, 0xEA, 0x43, 0x02, 0x6A, 0x50, 0x49 \}

**Listing 1: Contents of the test\_vcipher.c file**

```c
int main(){
/* Appendix B p.33 Round 1 */
vector unsigned char state = { 0x19, 0x3d, 0xE3, 0xBE, 0xA0, 0xF4, 0xE2, 0x2B, 0x9A, 0xC6, 0x8D, 0x2A, 0xE9, 0xF8, 0x48, 0x08 };;
vector unsigned char roundKey = { 0xA0, 0xFA, 0xFE, 0x17, 0x88, 0x54, 0x2C, 0xB1, 0x23, 0xA3, 0x39, 0x39, 0x2A, 0x6C, 0x76, 0x05 };;
vector unsigned char expect = { 0xA4, 0x9C, 0x7F, 0xF2, 0x68, 0x9F, 0x35, 0x2B, 0x6B, 0x5B, 0xEA, 0x43, 0x02, 0x6A, 0x50, 0x49 };;
vector unsigned char answer = __vcipher(state, roundKey);
return answer == expect ? 0 : 1;
}
```

**Example of \texttt{__vsbox}**

Again, the input is the intermediate block listed in Appendix B of \textit{Federal Information Processing Standards Publication 197 (FIPS-197)}.
Achieving high performance for Advanced Encryption Standard (AES) applications

Figure 9. SubByte operation example based on Appendix B of FIPS-197

At the beginning of round 1, the input is:

{ 0x19, 0x3d, 0xBE, 0xA0, 0xF4, 0xE2, 0xB2, 0x9A, 0xC6, 0x8D, 0x2A, 0xE9, 0xF8, 0x48, 0x08 }

After going through SubBytes(), the output, which is the result of the pre-defined S-box lookup will be:

{ 0xD4, 0x27, 0x11, 0xAE, 0xE0, 0xBF, 0x98, 0xF1, 0xB8, 0xB4, 0x5D, 0xE5, 0x1E, 0x52, 0x30 }

Listing 2: Contents of the test_vsbox.c file

```c
int main(){
/* Appendix B p.33 Round 1 */
    vector unsigned char state = { 0x19, 0x3d, 0xBE, 0xA0, 0xF4, 0xE2, 0xB2, 0x9A, 0xC6, 0x8D, 0x2A, 0xE9, 0xF8, 0x48, 0x08 };
    vector unsigned char expect = { 0xD4, 0x27, 0x11, 0xAE, 0xE0, 0xBF, 0x98, 0xF1, 0xB8, 0xB4, 0x5D, 0xE5, 0x1E, 0x41, 0x52, 0x30 };
    vector unsigned char answer = __vsbox(state);
    return answer == expect ? 0 : 1;
}
```

The code in Listing 1 and Listing 2 can be compiled with the following command:

```
xlc -o test_vsbox -qarch=pwr8 -qaltivec ./test_vsbox.c
```

When compiling at higher optimization levels, additional option `-qtune=pwr8` is recommended. The existence of POWER8 specific options `-qarch=pwr8` and `-qtune=pwr8`, controls the code generated by the compiler. They inform the compiler to adjust the instructions, scheduling, and other optimizations to achieve the best performance on POWER8 processor.

**Code generated by the compiler**

The following are two similar test programs, one using the POWER8 cryptography built-in function and one not.

Listing 3: Contents of the test_vsbox.c file, using built-in function

```c
int main(){
/* Appendix B p.33 Round 1 */
    vector unsigned char state = { 0x19, 0x3d, 0xBE, 0xA0, 0xF4, 0xE2, 0xB2, 0x9A, 0xC6, 0x8D, 0x2A, 0xE9, 0xF8, 0x48, 0x08 };
    vector unsigned char expect = { 0xD4, 0x27, 0x11, 0xAE, 0xE0, 0xBF, 0x98, 0xF1, 0xB8, 0xB4, 0x5D, 0xE5, 0x1E, 0x41, 0x52, 0x30 };
    vector unsigned char answer = __vsbox(state);
    return answer == expect ? 0 : 1;
}
```
Both programs can be compiled with `–c` and `–qlist` to generate the listing file (or alternatively, compiling them with `–c` and `–S` to generate the assembly files).

**Figure 10. Listing file when using built-in function**

The listing file of test__vcipher.c, the program with POWER8 built-in function, reveals that the program quickly terminates after the POWER8 instruction `vcipher` is called. The instruction count is 35.

On the other hand, the test program not using the POWER8 built-in functions is shown in Listing 4.

**Listing 4: Contents of file test_no__vsbox.c, without built-in function**
extern vector unsigned char mySubBytes(vector unsigned char);
extern vector unsigned char myShiftRows(vector unsigned char);
extern vector unsigned char myMixColumns(vector unsigned char);
extern vector unsigned char myAddRoundKey(vector unsigned char, vector unsigned char);

int main()
{ /* Appendix B p.33 Round 1 */
  vector unsigned char state
  = { 0x19, 0x3d, 0xE3, 0xBE, 0xA0, 0xF4, 0xE2, 0x2B, 0x9A, 0xC6, 0x8D, 0x2A, 0xE9, 0xF8, 0x48, 0x08 }; 
  vector unsigned char roundKey
  = { 0xA0, 0xFA, 0xFE, 0x17, 0x88, 0x54, 0x2C, 0xB1, 0x23, 0xA3, 0x39, 0x39, 0x2A, 0x6C, 0x76, 0x05 }; 
  vector unsigned char expect
  = { 0xA4, 0x9C, 0x7F, 0xF2, 0x68, 0x9F, 0x35, 0x2B, 0x6B, 0x5B, 0xEA, 0x43, 0x02, 0x6A, 0x50, 0x49 }; 
  vector unsigned char answer = mySubBytes(state);
  answer = myShiftRows(answer);
  answer = myMixColumns(answer);
  answer = myAddRoundKey(answer, roundKey);
  return answer == expect ? 0 : 1;
}

Figure 11. Listing file when not using built-in function

The listing file shows that the program makes four calls to the supporting functions. Function calls, together with the associated tasks to set up and manage the call stacks, are expensive in terms of performance cost. Even if the optimization decides to inline those function calls to improve performance, the inline code will have hundreds, if not thousands of instructions. Compared to the built-in functions, the user-defined functions require more time to run and a larger footprint for the generated binary.

Another practical reason that makes using AES built-in functions critical is the mode of operation implemented by the program. If the selected mode of operation is chaining mode (for example CBC or PCBC) or feedback mode (for example CFB or OFB), one or even both of the encryption and decryption cannot be parallelized. The sequential operations then become the bottleneck in terms of performance. This is what makes using built-in functions essential for high performance applications.
Resources

- Visit the XL C/C++ for Linux and XL C/C++ for AIX product pages for more information.
- Get the free trial download for XL C/C++ for Linux.
- Join the rational C/C++ Cafe community and get connected.

References


© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)