***Vectorization analysis for basic intrinsics***

The following analysis will focus on explaining what sections of encrypt/decrypt functions are suitable for vectorization and which are not. It is important to mention that here we are only considering basic intrinsics. In the next section specialized intrinsics for encryption/decryption will be presented and problems that here could not be solved will find a solution with just calling an instruction. For those functions that are suitable for vectorization, the code using sse3 instructions will be presented and the results of running the original and modified code as well.

According to the standard, the encryption/decryption process is carried out following a set of strict steps. Since modularity is one of the main features of the AES128 implementation presented, it is easy to follow the process step by step. The encryption/decryption functions can be considered as wrappers receiving one block at a time. The following programs show the functions called from the encryption/decryption wrappers.

***Encryption***

addRoundKey**(**block**,** **&**expandedKey**[(**keyIndex**++)** **\*** BLOCK\_SIZE**]);**

**for(**i **=** 0**;** i**<**9**;** i**++){**

subBytes**(**block**,** BLOCK\_SIZE**);**

shiftRows**(**block**);**

mixColumns**(**block**);**

addRoundKey**(**block**,** **&**expandedKey**[(**keyIndex**++)** **\*** BLOCK\_SIZE**]);**

**}**

subBytes**(**block**,** BLOCK\_SIZE**);**

shiftRows**(**block**);**

addRoundKey**(**block**,** **&**expandedKey**[(**keyIndex**++)** **\*** BLOCK\_SIZE**]);**

***Decryption***

addRoundKey**(**block**,** **&**expandedKey**[(**keyIndex**--)** **\*** BLOCK\_SIZE**]);**

**for(**i **=** 0**;** i **<** 9**;** i**++){**

invShiftRows**(**block**);**

invSubBytes**(**block**,** BLOCK\_SIZE**);**

addRoundKey**(**block**,** **&**expandedKey**[(**keyIndex**--)** **\*** BLOCK\_SIZE**]);**

invMixColumns**(**block**);**

**}**

invShiftRows**(**block**);**

invSubBytes**(**block**,** BLOCK\_SIZE**);**

addRoundKey**(**block**,** **&**expandedKey**[(**keyIndex**--)** **\*** BLOCK\_SIZE**]);**

**Byte Substitution**

As its name states, this function is used to interchange the bytes of a block with a predefined byte array. The following program shows how this process is carried out for both the regular and the inverse byte substitution.

const unsigned char subst\_array**[**256**]={**0x63**,**0x7C**,**0x77**,**0x7B**,**0xF2**, … ,**0x76**};**

void subBytes**(**unsigned char**\*** const block**,** const unsigned int len**){**

int i**;**

unsigned char**\*** ptr **=** block**;**

**for(**i **=** 0**;** i **<** len**;** i**++){**

**\***ptr **=** subst\_array**[\***ptr**];**

ptr**++;**

**}**

**}**

const unsigned char inv\_subst\_array**[**256**]={** 0x52**,** 0x09**,** 0x6A**, … ,**0x76**};**

void invSubBytes**(**unsigned char**\*** const block**,** const unsigned int len**){**

int i**;**

unsigned char**\*** ptr **=** block**;**

**for(**i **=** 0**;** i **<** len**;** i**++){**

**\***ptr **=** inv\_subst\_array**[\***ptr**];**

ptr**++;**

**}**

**}**

**}**

As it could be implied, the functions are storing in block[i] the contents of the substitution array in the location pointed by the current value of the block[i] position. Since we cannot be sure we are accessing contiguous locations in the substitution array, vectorization is not feasible in this case for both functions.

***Rows Shifting***

This functions operate on the rows of the state; it cyclically shifts the bytes in each row by a certain offset.

void shiftRows**(**unsigned char**\*** const block**){**

//Shift first row

temp **=** block**[**1**];**

block**[**1**]** **=** block**[**5**];**

block**[**5**]** **=** block**[**9**];**

block**[**9**]** **=** block**[**13**];**

block**[**13**]** **=** temp**;**

//Shift second row

temp **=** block**[**2**];**

block**[**2**]** **=** block**[**10**];**

void invShiftRows**(**unsigned char**\*** const block**){**

//Shift first row

temp **=** block**[**13**];**

block**[**13**]** **=** block**[**9**];**

block**[**9**]** **=** block**[**5**];**

block**[**5**]** **=** block**[**1**];**

block**[**1**]** **=** temp**;**

//Shift second row

temp **=** block**[**14**];**

block**[**14**]** **=** block**[**6**];**

block**[**6**]** **=** temp**;**

temp **=** block**[**10**];**

block**[**10**]** **=** block**[**2**];**

block**[**2**]** **=** temp**;**

//Shift third row

temp **=** block**[**7**];**

block**[**7**]** **=** block**[**11**];**

block**[**11**]** **=** block**[**15**];**

block**[**15**]** **=** block**[**3**];**

block**[**3**]** **=** temp**;**

**}**

block**[**10**]** **=** temp**;**

temp **=** block**[**6**];**

block**[**6**]** **=** block**[**14**];**

block**[**14**]** **=** temp**;**

//Shift third row

temp **=** block**[**3**];**

block**[**3**]** **=** block**[**15**];**

block**[**15**]** **=** block**[**11**];**

block**[**11**]** **=** block**[**7**];**

block**[**7**]** **=** temp**;**

**}**

If we check carefully the code, vectorization cannot be applied given that non-contiguous memory location are assigned to block. For example block [1] is assigned the value of block [5]. There is no lineal relation between the displacements.

***Mix Columns***

In the MixColumns step, the four bytes of column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with ShiftRows, MixColumns provides diffusion in the cipher.

void invMixColumns**(**unsigned char**\*** const block**){**

temp **=** block**;**

**for(**i **=** 0**;** i **<** 32**;** i**++){**

invMixColumn**(**temp**);**

temp **+=** WORD\_SIZE**;**

**}**

**}**

void invMixColumn**(**unsigned char**\*** const column**){**

**for(**i **=** 0**;** i**<32;** i**++){**

t**[**i**]** **=** column**[**i**];**

**}**

column**[**0**]** **=** x14**[**t**[**0**]]** **^** x11**[**t**[**1**]]** **^** x13**[**t**[**2**]]** **^** x9**[**t**[**3**]];**

column**[**1**]** **=** x9**[**t**[**0**]]** **^** x14**[**t**[**1**]]** **^** x11**[**t**[**2**]]** **^** x13**[**t**[**3**]];**

column**[**2**]** **=** x13**[**t**[**0**]]** **^** x9**[**t**[**1**]]** **^** x14**[**t**[**2**]]** **^** x11**[**t**[**3**]];**

column**[**3**]** **=** x11**[**t**[**0**]]** **^** x13**[**t**[**1**]]** **^** x9**[**t**[**2**]]** **^** x14**[**t**[**3**]];**

**}**

void mixColumns**(**unsigned char**\*** const block**){**

temp **=** block**;**

**for(**i **=** 0**;** i **<** WORD\_SIZE**;** i**++){**

mixColumn**(**temp**);**

temp **+=** WORD\_SIZE**;**

**}**

**}**

void mixColumn**(**unsigned char **\*** const column**){**

**for(**i **=** 0**;** i **<** WORD\_SIZE**;** i**++){**

t**[**i**]** **=** column**[**i**];**

**}**

column**[**0**]** **=** x2**[**t**[**0**]]** **^** x3**[**t**[**1**]]** **^** t**[**2**]** **^** t**[**3**];**

column**[**1**]** **=** t**[**0**]** **^** x2**[**t**[**1**]]** **^** x3**[**t**[**2**]]** **^** t**[**3**];**

column**[**2**]** **=** t**[**0**]** **^** t**[**1**]** **^** x2**[**t**[**2**]]** **^** x3**[**t**[**3**]];**

column**[**3**]** **=** x3**[**t**[**0**]]** **^** t**[**1**]** **^** t**[**2**]** **^** x2**[**t**[**3**]];**

**}**

const unsigned char x2**[]** **=** **{**0x00**,**0x02**,**0x04**,**0x06**,**0x08**,**…**,**0x0c**};**

const unsigned char x3**[]** **=** **{**0x00**,**0x03**,**0x06**,**0x05**,**0x0c**,**…**,**0x0a};

const unsigned char x9**[]** **=** **{**0x00**,**0x09**,**0x12**,**0x1b**,**0x24**,**…**,**0x3f**};**

const unsigned char x11**[]** **=** **{**0x00**,**0x0b**,**0x16**,**0x1d**,**0x2c**,**…**,**0x31**};**

const unsigned char x13**[]** **=** **{**0x00**,**0x0d**,**0x1a**,**0x17**,**0x34**,**…**,**0x23**};**

const unsigned char x14**[]** **=** **{**0x00**,**0x0e**,**0x1c**,**0x12**,**0x38**,**…**,**0x70};

Taking a look into the structure of the code, it is easy to see that vectorization is not feasible for these functions. The problem is that they are not accessing contiguous memory. Dependencies are not a problem in this case, so we conclude that maybe with a complex rearrangement vectorization could be possible for this functions. We let that possibility out of the scope of this work.

***Add Round Key***

In the AddRoundKey step, the subkey is combined with the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR. Following is the original code in our implementation.

void addRoundKey**(**unsigned char**\*** const block**,** const unsigned char**\*** const key**){**

unsigned char**\*** a **=** block**;**

const unsigned char**\*** b **=** key**;**

**for(**i **=** 0**;** i **<** BLOCK\_SIZE**;** i**++){**

**\***a **=** **(\***a**)^(\*(**b**++));**

a**++;**

**}**

**}**

The above code is yielding out for optimization. Since a bitwise xor is applied and the block size is 128 bits, the whole operation can be reduced to a vector xor on 16 words of 8 bits. Following is the modified code.

void addRoundKey**(**unsigned char**\*** const block**,** const unsigned char**\*** const key**){**

unsigned char**\*** a **=** block**;**

const unsigned char**\*** b **=** key**;**

\_\_m128i aVec**;**

\_\_m128i bVec**;**

\_\_m128i res**;**

aVec **=** \_mm\_load\_si128**((**\_\_m128i**\*)**a**);**

bVec **=** \_mm\_load\_si128**((**\_\_m128i**\*)**b**);**

res **=** \_mm\_xor\_si128 **(**aVec**,** bVec**);**

\_mm\_store\_si128**((**\_\_m128i**\*)**a**,** res**);**

**}**

The above code is loading the block corresponding to the state and key and then using an intrinsic for xor operation. At the end a store instrinsic is used to store the whole modified block.

In order to give proof of the improvement of using vectorization we ran both an encryption and decryption test with both the original and modified code. As test file we selected a file with around 10,000 lines. The system for testing was a machine with Ubuntu 14.04 OS and an Intel Core i5 processor of four cores. The results are shown in the following table.

|  |  |  |
| --- | --- | --- |
| Operation | Execution Time Original | Execution Time Vectorization |
| Encryption |  |  |
| Decryption |  |  |