Implementing an FIR Filter on the MPC55xx

by: Alistair Robertson
Freescale Semiconductor, Inc.

1 Introduction

The e200z3 and e200z6 based MPC55xx devices contain the signal processing extension (SPE) auxiliary processing unit (APU). The SPE-APU provides a set of single instruction multiple data (SIMD) instructions. The SIMD instructions typically involve performing the same operation on multiple data elements stored in a single 64-bit register. Through the implementation of SIMD instructions, including vector multiply and accumulate (MAC) instructions, the SPE-APU provides digital signal processing (DSP) functionality. This can be used to accelerate signal processing routines, such as finite impulse response (FIR), infinite impulse response (IIR), and discrete fourier transforms (DFT).

This application note uses the example of a low-order FIR filter to demonstrate how this can be encoded for optimal use of the SPE-APU. This introduces the SPE programmers interface model (PIM) and demonstrates the use of C intrinsics to encode functions.
Implementing the FIR Filter using the SPE-APU

This document also gives an example of encoding directly in SPE assembly language and highlights the benefits of each method. Performance data for the C-level intrinsic functions, the SPE assembly function, and a function encoded directly in standard C is provided for comparison.

2 Implementing the FIR Filter using the SPE-APU

The FIR filter can be represented in this form:

\[ y(n) = \sum_{k=0}^{N-1} h(k) \cdot x(n-k) \]

Where \( M \) is the number of taps, \( h(k) \) is a vector of coefficients and \( x(k) \) is the input vector. For example, the 20\(^{th} \) element of the output vector of a 10-TAP filter can be represented as:

\[ y(20) = h(0) \cdot x(20) + h(1) \cdot x(19) + h(2) \cdot x(18) \ldots \ldots + h(9) \cdot x(11) \]

Eqn. 1

Thus, each entry in the output vector is the accumulated product of \( N \) multiplications of a coefficient by the corresponding delayed data sample. This can be implemented efficiently using the vector MAC instructions provided by the SPE-APU.

The subsequent series of diagrams demonstrate how the data can be manipulated within the 64-bit general-purpose registers (GPRs). Figure 1 shows the coefficients \( h(0) \) through \( h(9) \) that are pre-loaded into the 64-bit GPRs. The first elements of the input vector \( x(n) \) are also loaded into the GPRs. Execution of the first MAC instruction results in a vector containing two elements resident in the 64-bit accumulator: \( h[9] \cdot x[0] \) and \( h[9] \cdot x[1] \).

The next MAC instruction multiplies the next vectors of data elements and odd coefficients and accumulate them with the previous results. Thus, with a single SPE-APU SIMD instruction, two MACs are executed per clock cycle.

To shift the data through the delay line, a series of vector merge instructions can be executed. This is shown in Figure 2, where the resultant vector is the offset of the original vector.

Figure 3 shows the final part of the calculation. Again, a series of MAC instructions are executed, this time multiplying the even coefficients with the input data elements. In this manner, two elements of the output vector are calculated in parallel.

To continue this process for the full vector of input data, the merge process is repeated, and another two elements of the input data vector (one vector load) are loaded.
Implementing the FIR Filter using the SPE-APU

Implementing an FIR Filter on the MPC55xx, Rev. 0

Freescale Semiconductor

Figure 1. Vector Multiply Accumulate Instructions

Figure 2. Execution of Vector Merge Instructions to Delay Data
3 Programming Using Intrinsics

Currently there are two methods to develop functions that use the SPE-APU. One method is to program directly using SPE assembly instructions. Another, easier method is to use the C-intrinsics defined in the SPE programmers’ interface model (PIM).

The SPE-PIM defines a set of intrinsics with approximately one intrinsic for each available SPE instruction (with some additional intrinsics for data and register manipulation). Each intrinsic acts as a function call that, when compiled, generates the SPE instruction it represents.

However, the actual register allocation is left to the compiler and the code generated using the SPE intrinsics is PowerPC EABI compliant; therefore, the use of the SPE-PIM to generate SPE code is considerably easier than programming directly in assembly.

However, because the compiler allocates register usage for each intrinsic used, this can lead to some inefficiencies. An intrinsic call does not strictly result in generation of a single assembly instruction when compiled. Additional instructions to load/store data may be added. Section 5, “Performance,” details the performance impact of the different programming techniques. Appendix A shows the assembly code generated when compiling intrinsics and illustrates how one intrinsic may results in several assembly instructions.
3.1 SPE Data Types

The data types introduced by the SPE programming model are listed in the following table. The type __ev64__ stands for embedded vector of data width 64 bits.

<table>
<thead>
<tr>
<th>New C/C++ Type</th>
<th>Interpretation of Contents</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ev64_u16</strong></td>
<td>4 unsigned 16-bit integers</td>
<td>0...65535</td>
</tr>
<tr>
<td><strong>ev64_s16</strong></td>
<td>4 signed 16-bit integers</td>
<td>–32768...32767</td>
</tr>
<tr>
<td><strong>ev64_u32</strong></td>
<td>2 unsigned 32-bit integers</td>
<td>0...2^32 – 1</td>
</tr>
<tr>
<td><strong>ev64_s32</strong></td>
<td>2 signed 32-bit integers</td>
<td>–2^31...2^31 – 1</td>
</tr>
<tr>
<td><strong>ev64_u64</strong></td>
<td>1 unsigned 64-bit integer</td>
<td>0...2^64 – 1</td>
</tr>
<tr>
<td><strong>ev64_s64</strong></td>
<td>1 signed 64-bit integer</td>
<td>–2^63...2^63 – 1</td>
</tr>
<tr>
<td><strong>ev64_fs</strong></td>
<td>2 floats</td>
<td>IEEE-754 single-precision values</td>
</tr>
<tr>
<td><strong>ev64_opaque</strong></td>
<td>any of the above</td>
<td>—</td>
</tr>
</tbody>
</table>

The __ev64_opaque__ data type is an opaque data type that can represent any of the specified __ev64_*__ data types, and is the type used in the FIR example.

3.2 Intrinsics

This list of intrinsics shows those used in the FIR filter example.

Intrinsic: __ev_mhossfa
Instruction: evmhossfa d,a,b
Description: Vector multiply halfwords, odd, signed, saturate, fractional to accumulator
Usage: d = __ev_mhossfa(a,b).

In the FIR filter example, using opaque data types:

Accumulating_Product = __ev_mhossfa (InputVector0, Coefficients10)

This intrinsic wrote the product to the accumulator, overwriting the accumulator contents. It does not accumulate the product. This saves having to clear the accumulator.
Implementing an FIR Filter on the MPC55xx, Rev. 0

Intrinsic: __ev_mhossfaaw
Instruction: evmhossfaaw d,a,b
Description: Vector multiply halfwords, odd, signed, saturate, fractional and accumulate into words
Usage: \( d = \text{__ev_mhossfaaw}(a,b) \).

In the FIR filter example, using opaque data types:

\( \text{Accumulating\_Product} = \text{__ev_mhossfaaw(InputVector1, Coefficients8)} \)

In this case, the resultant product is accumulated.

---

Intrinsic: __ev_mergeloHi
Instruction: evmergelohi d,a,b
Description: Vector merge low/high
Usage: \( d = \text{__ev_mergeloHi}(a,b) \)

In the FIR filter example, using opaque data types:

\( \text{InputVector0} = \text{__ev_mergeloHi(InputVector0, InputVector1)} \);
In addition to intrinsics explicitly defining individual SPE instructions, there exists a set of data manipulation intrinsics. These intrinsics act like functions with parameters that are passed by value. These intrinsics are classified into:

- **Create intrinsics**
  - These intrinsics create new generic 64-bit opaque data types from the given inputs passed by value. In the FIR example, the 64-bit opaque variable are initialized as:
    - `Coefficients1 = __ev_create_u32(h_ptr[0], h_ptr[0]);`

- **Get intrinsics**
  - These intrinsics allow the user to access data from within a specified location of the generic 64-bit opaque data type. In the FIR example, the upper and lower results within the 64-bit opaque variable `Accumulating_Product` are extracted as follows:
    - `y_ptr[n] = __ev_get_upper_u32(Accumulating_Product)`
    - `y_ptr[n+1] = __ev_get_lower_u32(Accumulating_Product)`

- **Set intrinsics**
  - These intrinsics provide the capability of setting values in a 64-bit opaque data type that the intrinsic or the user specifies.

- **Convert intrinsics**
  - These intrinsics convert a generic 64-bit opaque data type to a specific signed or unsigned integral

### 4 10-TAP FIR Filter Code

This code provides an example of a 10-TAP FIR filter operating on fixed point data.

```c
unsigned int SPE_Vector_FIR(int N, short int *x_ptr, short int *y_ptr, short int *h_ptr) {
__ev64_opaque__ InputVector0, InputVector1, InputVector2, InputVector3, InputVector4, InputVector5;
__ev64_opaque__ Coefficients1, Coefficients2, Coefficients3, Coefficients4, Coefficients5, Coefficients6, Coefficients7, Coefficients8, Coefficients9, Coefficients10;
__ev64_opaque__ Accumulating_Product;
int n = 0;

//Load the coefficients into registers
Coefficients1 = __ev_create_u32(h_ptr[0], h_ptr[0]);
Coefficients2 = __ev_create_u32(h_ptr[1], h_ptr[1]);
Coefficients3 = __ev_create_u32(h_ptr[2], h_ptr[2]);
Coefficients4 = __ev_create_u32(h_ptr[3], h_ptr[3]);
Coefficients5 = __ev_create_u32(h_ptr[4], h_ptr[4]);
Coefficients6 = __ev_create_u32(h_ptr[5], h_ptr[5]);
Coefficients7 = __ev_create_u32(h_ptr[6], h_ptr[6]);
Coefficients8 = __ev_create_u32(h_ptr[7], h_ptr[7]);
Coefficients9 = __ev_create_u32(h_ptr[8], h_ptr[8]);
Coefficients10 = __ev_create_u32(h_ptr[9], h_ptr[9]);

//Clear the accumulator
__ev_set_acc_u64(0);
```

Implementing an FIR Filter on the MPC55xx, Rev. 0
Performance

Performance measurement in terms of cycles count is provided for the 10-TAP FIR function written in SPE assembly, using SPE intrinsics and written directly in C.

The performance of the 10-TAP FIR functions written in SPE assembly, SPE intrinsics, and directly in C-code has been measured and the results are provided. Included in the measured timings are the overhead of the function call, which includes saving and restoring the context to the stack. The measurements were made using an MPC5554 with the cache enabled, the stack loaded into the cache, and the flash access configured for 128 MHz operation. The C-code was compiled using Wind River Diab compiler version 5.3.0.0 with no optimizations. The source code for all the functions is provided in Appendices A, B, and C. The output vector of the FIR filter was set to 256 entries.
6 Summary

Writing code using the SPE-PIM makes efficient, but not optimum, use of the SPE-APU. There is a trade-off between the complexities of writing directly in assembly language and the performance levels achievable. In either case, the performance improvements over writing functions directly in C code are significant.

<table>
<thead>
<tr>
<th>Table 2. 10-TAP FIR Filter Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIR Source Code</strong></td>
</tr>
<tr>
<td>SPE Assembly</td>
</tr>
<tr>
<td>SPE C intrinsics</td>
</tr>
<tr>
<td>C-Code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Summary of Coding Techniques for SPE Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Development Effort</strong></td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>Code Readability</td>
</tr>
<tr>
<td>Relative Performance</td>
</tr>
<tr>
<td>Code Portability</td>
</tr>
</tbody>
</table>
Appendix A  Assembly Generated by Intrinsics

These intrinsics show assembly generated. Within the main loop, most intrinsics result in a single instruction. Outside the loop, the use of __ev_create_intrinsics result in greater than one assembly instruction due to the necessity to load and shift data.

```c
unsigned int SPE_Vector_FIR(int N, short int *x_ptr, short int *y_ptr, short int *h_ptr) {
    __SPE_Vect.:mrr11,r1
    stwu    r1,-0xC0(r1)
    mflr    r0
    subi    r11,r11,0x90
    bl      0x40000EF0 l
    stw     r3,0x20(r1)
    mr      r30,r4
    mr      r29,r5
    mr      r28,r6

    __ev64_opaque__ InputVector0, InputVector1, InputVector2, InputVector3, InputVector4,
    InputVector5;

    __ev64_opaque__ Coefficients1, Coefficients2, Coefficients3, Coefficients4, Coefficients5,
    Coefficients6, Coefficients7, Coefficients8, Coefficients9, Coefficients10;

    __ev64_opaque__ Accumulating_Product;
    int n = 0;
    li      r21,0x0

    Coefficients1    = __ev_create_u32(h_ptr[0], h_ptr[0]);
    addi    r11,r1,0x8
    lha     r10,0x0(r28)
    lha     r9,0x0(r28)
    evmergel o r9,r10,r9
    evstddx  r9,r0,r11
    Coefficients2    = __ev_create_u32(h_ptr[1], h_ptr[1]);
    lha     r12,0x2(r28)
    lha     r11,0x2(r28)
    evmergel o r20,r12,r11
    Coefficients3    = __ev_create_u32(h_ptr[2], h_ptr[2]);
    lha     r10,0x4(r28)
    lha     r12,0x4(r28)
    evmergel o r19,r10,r12
    Coefficients4    = __ev_create_u32(h_ptr[3], h_ptr[3]);
    lha     r11,0x6(r28)
    lha     r10,0x6(r28)
    evmergel o r18,r11,r10
    Coefficients5    = __ev_create_u32(h_ptr[4], h_ptr[4]);
    lha     r12,0x8(r28)
    lha     r11,0x8(r28)
    evmergel o r17,r12,r11
    Coefficients6    = __ev_create_u32(h_ptr[5], h_ptr[5]);
    lha     r10,0x0A(r28)
    lha     r12,0x0A(r28)
    evmergel o r16,r10,r12
    Coefficients7    = __ev_create_u32(h_ptr[6], h_ptr[6]);
```

Implementing an FIR Filter on the MPC55xx, Rev. 0

Freescale Semiconductor
Implementing an FIR Filter on the MPC55xx, Rev. 0

Freescale Semiconductor

Summary

```
lha    r11,0x0C(r28)
lha    r10,0x0C(r28)
evmrgelo r15,r11,r10
Coefficients8 = __ev_create_u32(h_ptr[7], h_ptr[7]);
lha    r12,0x0E(r28)
lha    r11,0x0E(r28)
evmrgelo r14,r12,r11
Coefficients9 = __ev_create_u32(h_ptr[8], h_ptr[8]);
addi    r10,1,0x10
lha    r12,0x10(r28)
lha    r9,0x10(r28)
evmrgelo r9,r12,r9
evstddx  r9,r0,r10
Coefficients10 = __ev_create_u32(h_ptr[9], h_ptr[9]);
addi     r11,1,0x18
lha    r12,0x12(r28)
lha    r10,0x12(r28)
evmrgelo r10,r12,r10
evstddx  r10,r0,r11
__ev_set_acc_u64(0);
li       r7,0x0
li       r8,0x0
evmrgelo r8,r7,r8
evmsra   r8,r8

InputVector0 = __ev_create_u32(x_ptr[n+0], x_ptr[n+1]);
  slwi    r12,r21,0x1
  lhax    r12,r30,r12
  slwi    r11,r21,0x1
  add     r11,r30,r11
  lha     r11,0x2(r11)
evmrgelo r27,r12,r11
InputVector1 = __ev_create_u32(x_ptr[n+2], x_ptr[n+3]);
  slwi    r12,r21,0x1
  add     r12,r30,r12
  lha     r12,0x4(r12)
  slwi    r11,r21,0x1
  add     r11,r30,r11
  lha     r11,0x6(r11)
evmrgelo r26,r12,r11
InputVector2 = __ev_create_u32(x_ptr[n+4], x_ptr[n+5]);
  slwi    r12,r21,0x1
  add     r12,r30,r12
  lha     r12,0x8(r12)
  slwi    r11,r21,0x1
  add     r11,r30,r11
  lha     r11,0x10A(r11)
evmrgelo r25,r12,r11
InputVector3 = __ev_create_u32(x_ptr[n+6], x_ptr[n+7]);
  slwi    r12,r21,0x1
  add     r12,r30,r12
  lha     r12,0x0C(r12))
  slwi    r11,r21,0x1
  add     r11,r30,r11
  lha     r11,0x0E(r11)
evmrgelo r24,r12,r11
```
### Summary

```c
InputVector4 = __ev_create_u32(x_ptr[n+8], x_ptr[n+9])
    slwi  r12, r21, 0x1
    add   r12, r30, r12
    lha   r12, 0x10(r12)
    slwi  r11, r21, 0x1
    add   r11, r30, r11
    lha   r11, 0x12(r11)
    evmergelo r23, r12, r11

for (n=0; n<N; n+=2)
    li    r21, 0x0
    lwz   r12, 0x20(r1)
    cmpw  r21, r12
    bge   0x40000A30
{
    InputVector5 = __ev_create_u32(x_ptr[n+10], x_ptr[n+11])
        slwi  r12, r21, 0x1
        add   r12, r30, r12
        lha   r12, 0x14(r12)
        slwi  r11, r21, 0x1
        add   r11, r30, r11
        lha   r11, 0x16(r11)
        evmergelo r22, r12, r11

    Accumulating_Product = __ev_mhossfa(InputVector0, Coefficients10)
        addi  r12, r1, 0x18
        evlddx r12, r0, r12
        evmhossfa r31, r27, r12
    Accumulating_Product = __ev_mhossfaaw(InputVector1, Coefficients8)
        evmhossfaaw r31, r26, r14
    Accumulating_Product = __ev_mhossfaaw(InputVector2, Coefficients6)
        evmhossfaaw r31, r25, r16
    Accumulating_Product = __ev_mhossfaaw(InputVector3, Coefficients4)
        evmhossfaaw r31, r24, r18
    Accumulating_Product = __ev_mhossfaaw(InputVector4, Coefficients2)
        evmhossfaaw r31, r23, r20

    InputVector0 = __ev_mergelohi(InputVector0, InputVector1)
        evmergelohi r27, r27, r26
    InputVector1 = __ev_mergelohi(InputVector1, InputVector2)
        evmergelohi r26, r26, r25
    InputVector2 = __ev_mergelohi(InputVector2, InputVector3)
        evmergelohi r25, r25, r24
    InputVector3 = __ev_mergelohi(InputVector3, InputVector4)
        evmergelohi r24, r24, r23
    InputVector4 = __ev_mergelohi(InputVector4, InputVector5)
        evmergelohi r23, r23, r22

    Accumulating_Product = __ev_mhossfaaw(InputVector0, Coefficients9)
        addi  r10, r1, 0x10
        evlddx r10, r0, r10
        evmhossfaaw r31, r27, r10
    Accumulating_Product = __ev_mhossfaaw(InputVector1, Coefficients7)
        evmhossfaaw r31, r26, r15
    Accumulating_Product = __ev_mhossfaaw(InputVector2, Coefficients5)
        evmhossfaaw r31, r25, r17
    Accumulating_Product = __ev_mhossfaaw(InputVector3, Coefficients3)
```

Implementing an FIR Filter on the MPC55xx, Rev. 0

Freescale Semiconductor
evmhossfaaw r31,r24,r19
Accumulating_Product = __ev_mhossfaaw(InputVector4, Coefficients1);
addi r10,r1,0x8
evlddx r10,r0,r10
evmhossfaaw r31,r23,r10

y_ptr[n] = __ev_get_upper_u32(Accumulating_Product);
slwi r12,r21,0x1
evmergehilo r11,r31,r31
sthx r11,r29,r12
y_ptr[n+1] = __ev_get_lower_u32(Accumulating_Product);
slwi r9,r21,0x1
add r9,r29,r9
evmergehilo r12,r31,r31
sth r12,0x2(r9)

InputVector0 = __ev_mergehilo(InputVector0, InputVector1);
evmergehilo r27,r27,r26
InputVector1 = __ev_mergehilo(InputVector1, InputVector2);
evmergehilo r26,r26,r25
InputVector2 = __ev_mergehilo(InputVector2, InputVector3);
evmergehilo r25,r25,r24
InputVector3 = __ev_mergehilo(InputVector3, InputVector4);
evmergehilo r24,r24,r23
InputVector4 = InputVector5;
evmergehilo r23,r22,r22
addi r21,r21,0x2

);}  
  b 0x4000097C
);}
Appendix B  FIR Filter Written in Assembly Code

FIR filter written directly in assembly code, excluding context save and restore routine.

#----------------------------------------------------
# Load coeffs into registers
#----------------------------------------------------
evwlhou    r16, 0(r6);  # r16 = Coefficient[1]
evwlhou    r17, 4(r6);  # r17 = Coefficient[2]
evwlhou    r18, 8(r6);  # r18 = Coefficient[3]
evwlhou    r19, 12(r6); # r19 = Coefficient[4]
evwlhou    r20, 16(r6); # r12 = Coefficient[5]
evwlhou    r21, 20(r6); # r21 = Coefficient[6]
evwlhou    r22, 24(r6); # r21 = Coefficient[7]
evwlhou    r23, 28(r6); # r23 = Coefficient[8]
evwlhou    r24, 32(r6); # r24 = Coefficient[9]
evwlhou    r25, 36(r6); # r25 = Coefficient[10]

#----------------------------------------------------
# Initialize counter to zero
#----------------------------------------------------
li r15, 0x0  # use r15 as a counter

#----------------------------------------------------
# Load first set of data to be processed
#----------------------------------------------------
evwlhou    r26, 0(r4); # Load InputVector_0
(evwlhou    r27, 4(r4); # Load InputVector_1
(evwlhou    r28, 8(r4); # Load InputVector_2
(evwlhou    r29, 12(r4); # Load InputVector_3
(evwlhou    r30, 16(r4); # Load InputVector_4

#----------------------------------------------------
# Loop
#----------------------------------------------------
loop:
evwlhou    r31, 20(r4); # Load InputVector_5
evmhossfa  r6, r26, r25; # InputVector_0 * Coefficients 10
evmhossfaaw r6, r27, r23; # InputVector_1 * Coefficients 8
evmhossfaaw r6, r28, r21; # InputVector_2 * Coefficients 6
evmhossfaaw r6, r29, r19; # InputVector_3 * Coefficients 4
evmhossfaaw r6, r30, r17; # InputVector_4 * Coefficients 2

evmergelohi r26, r26, r27; # InputVector0 = Merge InputVector_0 & InputVector_1
evmergelohi r27, r27, r28; # InputVector1 = Merge InputVector_1 & InputVector_2
evmergelohi r28, r28, r29; # InputVector2 = Merge InputVector_2 & InputVector_3
evmergelohi r29, r29, r30; # InputVector3 = Merge InputVector_3 & InputVector_4
evmergelohi r30, r30, r31; # InputVector4 = Merge InputVector_4 & InputVector_5
evmhossfaaw r6, r26, r24; # InputVector_0 * Coefficients 9
evmhossfaaw r6, r27, r22; # InputVector_1 * Coefficients 7
evmhossfaaw r6, r28, r20; # InputVector_2 * Coefficients 5
evmhossfaaw r6, r29, r18; # InputVector_3 * Coefficients 3
evmhossfaaw r6, r30, r16; # InputVector_4 * Coefficients 1
# Store output
    evstwho        r6, 0(r5); # Update pointer to output vector for next results
    addi           r5, r5, 4;

    evmergelohi    r26, r26, r27; # InputVector0  = Merge InputVector_0 & InputVector_1
    evmergelohi    r27, r27, r28; # InputVector1  = Merge InputVector_1 & InputVector_2
    evmergelohi    r28, r28, r29; # InputVector2  = Merge InputVector_2 & InputVector_3
    evmergelohi    r29, r29, r30; # InputVector3  = Merge InputVector_3 & InputVector_4
    evmergehilo    r30, r31, r31; # InputVector4  = InputVector_5

    addi           r4, r4, 4; # Update r4 to point to next input vector
    addi           r15, r15, 2; # Update counter 2 points
    cmpw           r15, r3;  # check if loop is done
    bne            loop;
Appendix C  FIR Filter Written in C Code

10-TAP FIR filter written in C-code for performance comparison.

```c
int fir10(int N, int *x_ptr, short int *y_ptr, int *h_ptr)
{
    int m, n, accumulator = 0;
    for (n = 0; n<N; n++)
    {
        for (m = 0; m<10; m++)
        {
            accumulator = accumulator + (*(h_ptr + m) * *(x_ptr + m));
        }
        *(y_ptr+n) = accumulator;
        accumulator = 0;
        x_ptr = x_ptr + 1;
    }
}
```