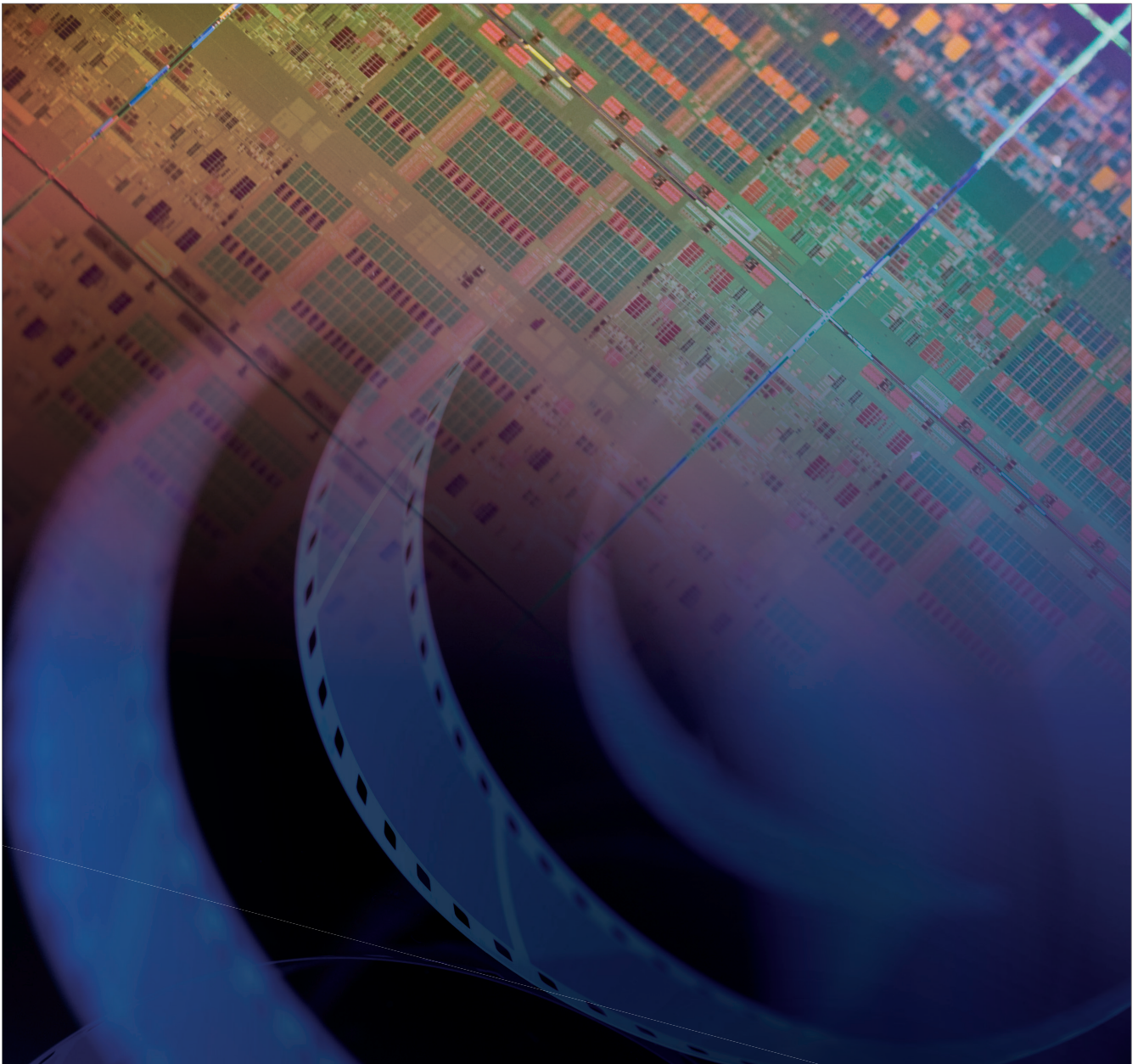# MIPS SIMD programming

Optimizing multimedia codecs

Introduction

The exponential growth of internet and mobile communication is driving the ever-increasing use of multimedia applications.

With the need for low channel bandwidth, limited storage capacity and low cost in today's mobile and cloud-based applications, there is a requirement for high quality audio and video at low bit rates. This has given rise to highly-efficient compression codecs such as VP9, HEVC and others, which involve a large amount of processing and complex coding tools. It is also critical that multimedia codecs are aggressively optimized to meet the needs of low power and cost-sensitive devices.

The MSA (MIPS SIMD Architecture) is designed to meet these requirements for multimedia and other compute-intensive applications. This paper demonstrates the use of the MSA in optimizing video codecs, and highlights other benefits of the MSA such as flexibility, programmer friendliness, 'future proof coding' and scalability.

Simple C programming reduces the time to market, facilitates faster upgrades and provides easy maintenance. Our simple but superior architecture and smart compiler technology leads to fast running, maintainable and portable codes in a shorter development time.

Video codecs and SIMD

As you can see from Figure 1, most of the modules in a typical video decoder can be optimized using SIMD as it involves processing multiple pixels simultaneously.

Video compression has a typical pixel depth of 8-bits or 10-bits; further mathematical operations can take intermediate results to 16-bits or 32-bits. Hence with an implementation of a typical 128-bit vector register size SIMD processor, it is possible to make a four to eight time reduction to mathematical and data load and store operations.

Due to the inherent scalability, the vectorization will adapt to process the required pixels simultaneously, as the register width changes.
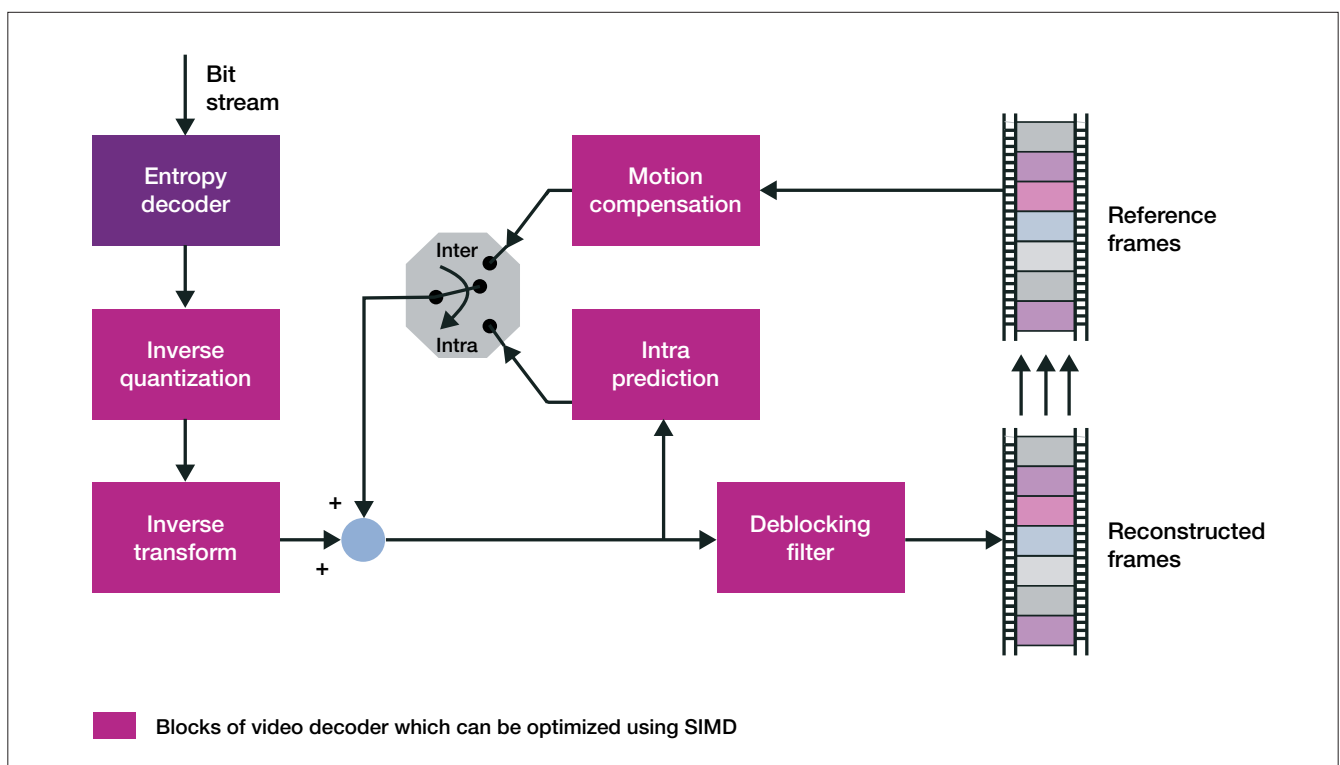

Figure 1: Block diagram of typical video decoder

MSA programming features

Frequently used operations in multimedia processing which can be vectorized using MSA include:

- Addition and subtraction operations
- Multiply and accumulate operations (dot product and simple multiplications)
- Logical and arithmetic shift operations (optional rounding)
- Other logical operations (AND,OR, XOR, etc.)
- Conditional selection or masking
- Load and store
- Pack-unpack and interleaving operations

Salient features of the MSA toolchain for fast and efficient programming include:

- Built-in intrinsics and data-types for all vector instructions usable in C and C++ programming
- Simple syntax for common operations (+, -, * operators can be used on vector data-types)
- Complete replacement for hand-coded assembly
- Auto scheduling by compiler
- Faster time-to-market, easier maintenance and quicker upgrades

The following two sections will demonstrate each of the above key features of the MSA using an example of a motion compensation filter in one of the latest video codec standards by Google – VP9.

MSA approach for optimizing video codecs

The VP9 codec, part of the Google-sponsored WebM open web media project, is a high-performance codec that compresses video files and streams to approximately half the size of previous generation encoding technologies, making it ideal for addressing HD video. VP9 will be used for YouTube and Google Hangouts as well as other web-based video applications.

VP9, with typical video compression blocks and added complexities for better compression, is a good example for explaining a typical approach to optimizing any video codec using the MSA.

Following are the steps taken in optimizing the VP9 decoder on the MSA:

- Identify compute-intensive modules
- Identify the compute-intensive functions which can be vectorized
- Evaluate input, output and intermediate data bit depths
- Evaluate a design strategy to combine similar operations on multiple data
- Implement the design strategy in MSA-C using appropriate vector data types and intrinsics

The following section illustrates this with an example of VP9 motion compensation vertical filtering.

VP9 motion compensation filter using MSA

Description

The example we are discussing is basically a Finite Impulse Response (FIR) filter that is used in VP9 for motion compensation. It is an 8-tap filter applied vertically over the image to derive sub-pixel rows.

- Read 8 pixels vertically (p0 to p7)
- Multiply and accumulate each pixel with corresponding filter-tap coefficient
- Round and divide by total filter weight.
- Clip the output in pixel range and write to output buffer
- Move the sliding window by one pixel position (p1=p0……p6=p7)
- Read new pixel position p7
- Repeat the above steps to motion compensate the block height

Code

Plain C code

Figure 2 on the following page shows an example of general purpose C code for the 8-tap Motion Compensation (MC) filter in VP9.

You will see that this code processes an 8-tap convolution (FIR) over a block of 16 times the number of H pixels where H represents the variable height of the block.

```c
#define ROUND_POWER_OF_TWO(value, n)  (((value) + (1 << ((n) - 1))) >> (n))

static inline unsigned char clip_pixel(int i32Val)
{
    return ((i32Val) > 255) ? 255u : ((i32Val) < 0) ? 0u : (i32Val);
}
void vert_filter_8taps_16width_c(unsigned char   *pSrc,       // SOURCE POINTER
                                 int              SrcStride,  // SOURCE BUFFER PITCH
                                 unsigned char   *pDst,       // DEST POINTER
                                 int              DstStride,  // DEST BUFFER PITCH
                                 char            *pFilter,    // POINTER TO FILTER BANK
                                 int              Height)     // HEIGHT OF THE BLOCK
{
    unsigned int Row, Col;
    int          FiltSum;
    short        Src0, Src1, Src2, Src3, Src4, Src5, Src6, Src7;

    pSrc -= (8 / 2 - 1) * SrcStride;  // MOVE INPUT SRC POINTER TO APPROPRIATE POSITION

    // LOOP FOR NUMBER OF COLUMNS-16
    for (Col = 0; Col < 16; ++Col)
    {
        Src0 = pSrc[0 * SrcStride];
        Src1 = pSrc[1 * SrcStride];
        Src2 = pSrc[2 * SrcStride];
        Src3 = pSrc[3 * SrcStride];
        Src4 = pSrc[4 * SrcStride];
        Src5 = pSrc[5 * SrcStride];
        Src6 = pSrc[6 * SrcStride];

        // LOOP FOR NUMBER OF ROWS
        for (Row = 0; Row < Height; Row++)
        {
            Src7 = pSrc[(7 + Row) * SrcStride];

            FiltSum = 0;
            // ACCUMULATED FILTER SUM += PIXEL * FILTER COEFF
            FiltSum += (Src0 * pi8Filter[0]);
            FiltSum += (Src1 * pi8Filter[1]);
            FiltSum += (Src2 * pi8Filter[2]);
            FiltSum += (Src3 * pi8Filter[3]);
            FiltSum += (Src4 * pi8Filter[4]);
            FiltSum += (Src5 * pi8Filter[5]);
            FiltSum += (Src6 * pi8Filter[6]);
            FiltSum += (Src7 * pi8Filter[7]);

            FiltSum = ROUND_POWER_OF_TWO(FiltSum, 7); // ROUNDING
            pDst[Row * DstStride] = clip_pixel(FiltSum);// CLIP RESULT IN 0-255(UNSIGNED CHAR)

            // PREPARING FOR NEXT CONVOLUTION- SLIDING WINDOW
            Src0 = Src1;
            Src1 = Src2;
            Src2 = Src3;
            Src3 = Src4;
            Src4 = Src5;
            Src5 = Src6;
            Src6 = Src7;
        }
        pSrc += 1;
        pDst += 1;
    }
}
```
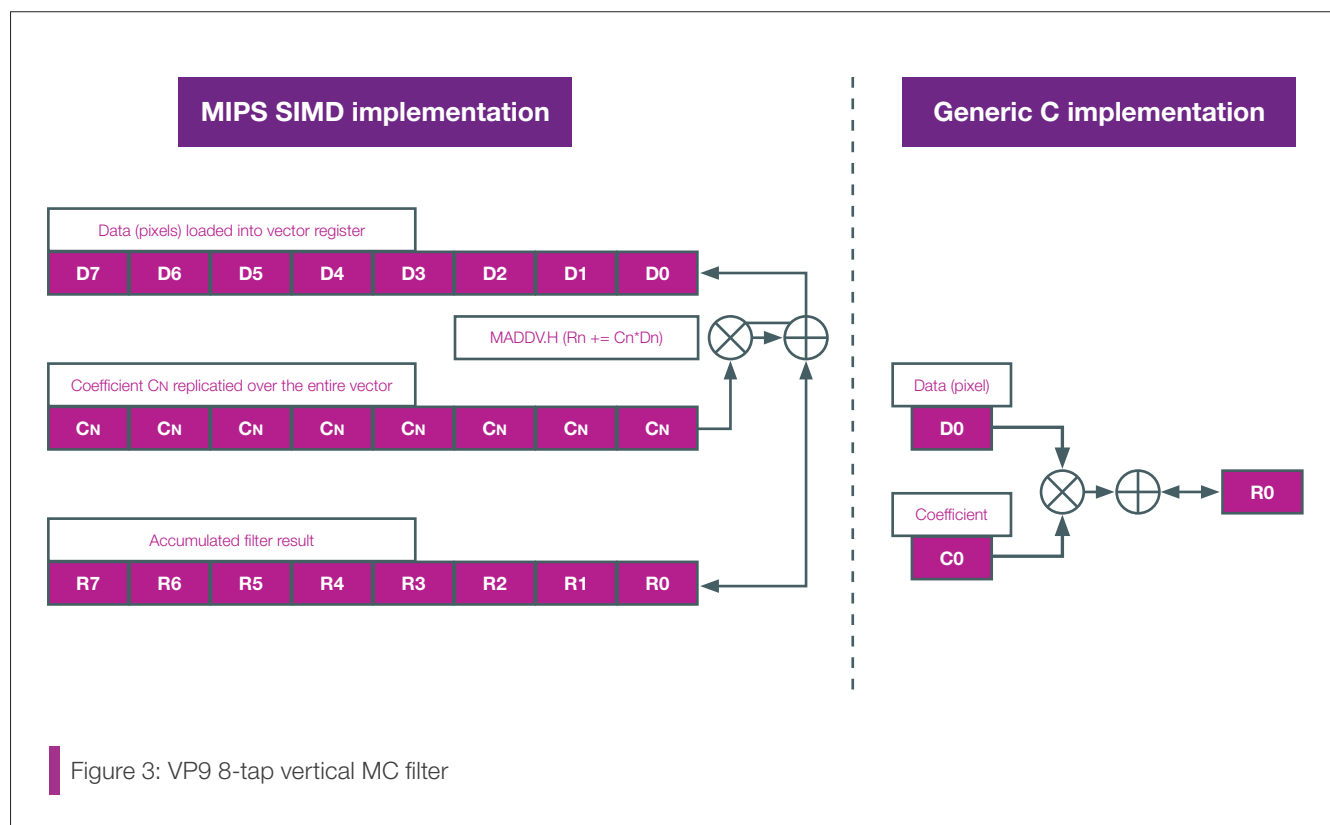
Figure 2: Example of general purpose C code for the 8-tap Motion Compensation (MC) filter in VP9

SIMD approach

The steps required for the SIMD approach are similar to the steps described for the VP9 motion compensation (see page 3), except a vector data type of 8 pixels is used instead of a single pixel operation.

The following figure demonstrates a SIMD optimized version of the core convolution code:

MIPS SIMD implementation

Data (pixels) loaded into vector register

D7	D6	D5	D4	D3	D2	D1	D0

MADDV.H (Rn += Cn*Dn)

Coefficient CN replicatied over the entire vector

CN	CN	CN	CN	CN	CN	CN	CN

Accumulated filter result

R7	R6	R5	R4	R3	R2	R1	R0

Generic C implementation

Data (pixel)

D0

Coefficient

C0

R0

Figure 3: VP9 8-tap vertical MC filter

MSA C code

```
/* MSA VECTOR TYPES */
#define WRLEN               128 // VECTOR REGISTER LENGTH 128-BIT
#define NUMWRELEM           (WRLEN >> 3)

typedef signed char IMG_VINT8 __attribute__ ((vector_size(NUMWRELEM))); //VEC_SIGNED_BYTES
typedef unsigned char IMG_VUINT8 __attribute__ ((vector_size(NUMWRELEM)));//VEC_UNSIGNED_BYTES
typedef short IMG_VINT16 __attribute__ ((vector_size(NUMWRELEM))); //VEC_SIGNED_HALF-WORD

#define LOAD_UNPACK_VEC(pSrc, SrcStride, vi16VecRight, vi16VecLeft)     \
{                                                                      \
    IMG_VUINT8  vu8Src;                                               \
    IMG_VINT16  vi16Vec0;                                             \
    IMG_VINT8   vi8Tmp0;                                              \
    /* LOAD INPUT VECTOR */                                           \
    vu8Src = *((IMG_VINT8 *)(pSrc));                                  \
    /* RANGE WARPING TO MAINTAIN 16 BIT PRECISION */                 \
    vi16Vec0 = __builtin_msa_xori_b(vu8Src, 128);                    \
    /* CALCULATE SIGN EXTENSION */                                    \
    vi8Tmp0 = __builtin_msa_clti_s_b(vi16Vec0, 0);                   \
    /* INTERLEAVE RIGHT TO 16 BIT VEC */                             \
    vi16VecRight = __builtin_msa_ilvr_b(vi8Tmp0, vi16Vec0);         \
    /* INTERLEAVE LEFT TO 16 BIT VEC */                              \
    vi16VecLeft = __builtin_msa_ilvl_b(vi8Tmp0, vi16Vec0);         \
    pSrc += SrcStride;                                               \
}

void vert_filter_8taps_16width_msa(unsigned char   *pSrc,       // SOURCE POINTER
                               int          SrcStride, // SOURCE BUFFER PITCH
                               unsigned char *pDst,     // DEST POINTER
                               int          DstStride, // DEST BUFFER PITH
                               char         *pFilter,  // POINTER TO FILTER BANK
                               int          Height)    // HEIGHT OF THE BLOCK
{
    int     u32LoopCnt;
    VINT16  vi16Vec0Right, vi16Vec1Right, vi16Vec2Right, vi16Vec3Right;
    VINT16  vi16Vec4Right, vi16Vec5Right, vi16Vec6Right, vi16Vec7Right;
    VINT16  vi16Vec0Left, vi16Vec1Left, vi16Vec2Left, vi16Vec3Left;
    VINT16  vi16Vec4Left, vi16Vec5Left, vi16Vec6Left, vi16Vec7Left;
    VINT16  vi16Temp1Right, vi16Temp1Left;
    VINT16  vi16Filt0, vi16Filt1, vi16Filt2, vi16Filt3;
    VINT16  vi16Filt4, vi16Filt5, vi16Filt6, vi16Filt7;

    pSrc -= (3 * SrcStride);

    // PREPARE FILTER COEFF IN VEC REGISTERS
    vi16Filt0 = __builtin_msa_fill_h(*(pFilter));
    vi16Filt1 = __builtin_msa_fill_h(*(pFilter + 1));
    vi16Filt2 = __builtin_msa_fill_h(*(pFilter + 2));
    vi16Filt3 = __builtin_msa_fill_h(*(pFilter + 3));
    vi16Filt4 = __builtin_msa_fill_h(*(pFilter + 4));
    vi16Filt5 = __builtin_msa_fill_h(*(pFilter + 5));
    vi16Filt6 = __builtin_msa_fill_h(*(pFilter + 6));
    vi16Filt7 = __builtin_msa_fill_h(*(pFilter + 7));

    //LOAD 7 INPUT VECTORS
    LOAD_UNPACK_VEC(pSrc, SrcStride, vi16Vec0Right, vi16Vec0Left)
    LOAD_UNPACK_VEC(pSrc, SrcStride, vi16Vec1Right, vi16Vec1Left)
    LOAD_UNPACK_VEC(pSrc, SrcStride, vi16Vec2Right, vi16Vec2Left)
    LOAD_UNPACK_VEC(pSrc, SrcStride, vi16Vec3Right, vi16Vec3Left)
    LOAD_UNPACK_VEC(pSrc, SrcStride, vi16Vec4Right, vi16Vec4Left)
    LOAD_UNPACK_VEC(pSrc, SrcStride, vi16Vec5Right, vi16Vec5Left)
    LOAD_UNPACK_VEC(pSrc, SrcStride, vi16Vec6Right, vi16Vec6Left)

    // START CONVOLUTION VERTICALLY
    for (u32LoopCnt = Height; u32LoopCnt--; )
    {
```

Figure 4: Example MSA C code
(continued on next page)

```
        //LOAD 8TH INPUT VECTOR
        LOAD_UNPACK_VEC(pSrc, SrcStride, vi16Vec7Right, vi16Vec7Left)

        /* FILTER CALC */
        IMG_VINT16  vi16Tmp1, vi16Tmp2;
        IMG_VINT8   vi8Tmp3;

        // 8 TAP VECTORIZED CONVOLUTION FOR RIGHT HALF
        vi16Tmp1 =  (vi16Vec0Right * vi16Filt0);
        vi16Tmp1 += (vi16Vec1Right * vi16Filt1);
        vi16Tmp1 += (vi16Vec2Right * vi16Filt2);
        vi16Tmp1 += (vi16Vec3Right * vi16Filt3);
        vi16Tmp2 =  (vi16Vec4Right * vi16Filt4);
        vi16Tmp2 += (vi16Vec5Right * vi16Filt5);
        vi16Tmp2 += (vi16Vec6Right * vi16Filt6);
        vi16Tmp2 += (vi16Vec7Right * vi16Filt7);
        vi16Temp1Right = __builtin_msa_adds_s_h(vi16Tmp1, vi16Tmp2);

        // 8 TAP VECTORIZED CONVOLUTION FOR LEFT HALF
        vi16Tmp1 =  (vi16Vec0Left * vi16Filt0);
        vi16Tmp1 += (vi16Vec1Left * vi16Filt1);
        vi16Tmp1 += (vi16Vec2Left * vi16Filt2);
        vi16Tmp1 += (vi16Vec3Left * vi16Filt3);
        vi16Tmp2 =  (vi16Vec4Left * vi16Filt4);
        vi16Tmp2 += (vi16Vec5Left * vi16Filt5);
        vi16Tmp2 += (vi16Vec6Left * vi16Filt6);
        vi16Tmp2 += (vi16Vec7Left * vi16Filt7);
        vi16Temp1Left = __builtin_msa_adds_s_h(vi16Tmp1, vi16Tmp2);

        // ROUNDING RIGHT SHIFT RANGE CLIPPING AND NARROWING
        vi16Temp1Right = __builtin_msa_srari_h(vi16Temp1Right, 7);
        vi16Temp1Right = __builtin_msa_sat_s_h(vi16Temp1Right, 7);
        vi16Temp1Left = __builtin_msa_srari_h(vi16Temp1Left, 7);
        vi16Temp1Left = __builtin_msa_sat_s_h(vi16Temp1Left, 7);
        vi8Tmp3 = __builtin_msa_pckev_b(vi16Temp1Left, vi16Temp1Right);
        vi8Tmp3 = __builtin_msa_xori_b(vi8Tmp3, 128);

        // STORE OUTPUT VEC
        *((IMG_VINT8 *)(pDst)) = (vi8Tmp3);

        pDst += DstStride;

        // PREPARING FOR NEXT CONVOLUTION- SLIDING WINDOW
        vi16Vec0Right = vi16Vec1Right;
        vi16Vec1Right = vi16Vec2Right;
        vi16Vec2Right = vi16Vec3Right;
        vi16Vec3Right = vi16Vec4Right;
        vi16Vec4Right = vi16Vec5Right;
        vi16Vec5Right = vi16Vec6Right;
        vi16Vec6Right = vi16Vec7Right;

        vi16Vec0Left = vi16Vec1Left;
        vi16Vec1Left = vi16Vec2Left;
        vi16Vec2Left = vi16Vec3Left;
        vi16Vec3Left = vi16Vec4Left;
        vi16Vec4Left = vi16Vec5Left;
        vi16Vec5Left = vi16Vec6Left;
        vi16Vec6Left = vi16Vec7Left;
    }
}
```

Figure 4: Example MSA C code
(continued)

```
    vi16Tmp1 += (vi16Vec1Left * vi16Filt1); GENERATES FOLLOWING ASSEMBLY
    MADDV.H   w4 w31 w29 // VECTOR MULTIPLY AND ACCUMULATE IN DEST.

    vu8Src = *((IMG_VINT8 *)(pu8Src)); GENERATES FOLLOWING ASSEMBLY
    LD.B      w1 0($4) // VECTOR LOAD
```

Figure 5: Using generic ' * ' operators on two vector variables to complete
multiplication operation

Analysis

MSA specific compiler friendly C code

The use of built-in data types and intrinsics makes C code quickly portable across all MSA core implementations, and hence developers need not worry about a particular MSA core implementation and its subtle properties.

The use of built-in data types and built-in intrinsics also indirectly instructs the compiler to make the best use of the SIMD instructions and architecture. The compiler will then efficiently use the available number of vector registers and instruction throughputs to generate the best possible code.

Use of simple syntax for common operations

In Figure 5 on the previous page, you can see that the multiplication operation has been done using generic ' * ' operators on two vector variables. Similarly, it is also evident that this can be used for load/store and other simple operations.

Maintain a readable order of processing

Programmers can maintain a readable order of processing as far as possible while the compiler takes care of efficiently rearranging or scheduling the instructions as per latencies.

Complete replacement for hand-coded assembly

- No manual register management is required, as the efficient code is produced by the complier itself, eliminating the need to depend on hand-written assembly, manual register management and other time consuming tasks.

- All instructions can be generated using built-ins and intrinsics.

- As use of MSA is not dependent on hand-crafted assembly and varying assembler syntax, this will give faster portability across platform specific compilers for operating systems like iOS, Android etc.

Quick time-to-market, easy maintenance and faster upgrades

The use of MSA built-in data types and intrinsics in C code enhances the ability to generate MSA optimizations in a faster turnaround time, unlike competing processors. It also enhances the maintainability, the ability to upgrade and the portability of the code to any future MSA architecture. This is ideal for the implementation of evolving standards and rapidly varying functionality, so that modifications and bug fixing can be done quickly.

Performance measurement

In the above example of an MC 8-tap vertical filter, the total instruction count for the plain C code is 10310 instructions compared to 850 instructions for MSA optimized C code.

Summary

Simply writing C code using MSA-built-ins and data-types yields fast-running, maintainable and portable code that can be created in a shorter development time. Meeting the real-time targets for compute-intensive applications like video codecs on low-power and cost-sensitive devices is easier to achieve using the MSA.

Imagination

enquiries@imgtec.com
www.imgtec.com

UK t: +44 1923 260511
US t: +1 408 530 5000