RISC-V Vector Performance Analysis

Guy Lemieux
CEO at VectorBlox Computing
Prof. at University of British Columbia

Collaborators:
Fredy Augusto Alves
José Augusto Nacif
Science and Technology Institute
Universidade Federal de Viçosa, Brazil

Motivation

RISC-V Vectors (RVV) is a highly anticipated extension

Very early performance analysis

- Vector spec is incomplete
- Give feedback on vector spec

Compare to

- Fixed-width SIMD
- Alternative vector architecture MXP

Find cause of performance differences

SIMD Instructions Considered Harmful

by David Patterson and Andrew Waterman on Sep 18, 2017 | Tags: Architecture, CPU, ISA, Parallelism

Example: SIMD vs Vector

```
void daxpy(size_t n, double a, const double x[], double y[])
{
 for (size_t i = 0; i < n; i++) {
   y[i] = a*x[i] + y[i];
 }
}
```

Example: SIMD vs Vector

```c
void daxpy(size_t n, double a, const double x[], double y[])
{
 for (size_t i = 0; i < n; i++) {
   y[i] = a*x[i] + y[i];
 }
}
```

MIPS (MSA)

```
# a0 is n, a2 is pointer to x[0], a3 is pointer to y[0], $w13 is a
  0:  li       a1,-2
  4:  and      a1,a0,a1          # a1 = floor(n/2)*2 (mask bit 0)
  8:  sll      t0,a1,0x3         # t0 = byte address of a1
  c:  addu     v1,a3,t0          # v1 = &y[a1]
 10:  beq      a3,v1,38          # if y==&y[a1] goto Fringe
                                 #    (t0==0 so n is 0 | 1)
 14:  move     v0,a2             # (delay slot) v0 = &x[0]
 18:  splati.d $w2,$w13[0]       # w2 = fill SIMD reg. with copies of a
Main Loop:
 1c:  ld.d     $w0,0(a3)         # w0 = 2 elements of y
 20:  addiu    a3,a3,16          # incr. pointer to y by 2 FP numbers
 24:  ld.d     $w1,0(v0)         # w1 = 2 elements of x
 28:  addiu    v0,v0,16          # incr. pointer to x by 2 FP numbers
 2c:  fmadd.d  $w0,$w1,$w2       # w0 = w0 + w1 * w2
 30:  bne      v1,a3,1c          # if (end of y != ptr to y) go to Loop
 34:  st.d     $w0,-16(a3)       # (delay slot) store 2 elts of y
Fringe:
 38:  beq      a1,a0,50          # if (n is even) goto Done
 3c:  addu     a2,a2,t0          # (delay slot) a2 = &x[n-1]
 40:  ldc1     $f1,0(v1)         # f1 = y[n-1]
 44:  ldc1     $f0,0(a2)         # f0 = x[n-1]
 48:  madd.d   $f13,$f1,$f13,$f0# f13 = f1+f0*f13 (muladd if n is odd)
 4c:  sdc1     $f13,0(v1)        # y[n-1] = f13 (store odd result)
Done:
 50:  jr      ra                 # return
 54:  nop                        # (delay slot)
```

Intel (AVX2)

```
# eax is i, n is esi, a is xmm1,
# pointer to x[0] is ebx, pointer to y[0] is ec
 0: push   esi
 1: push   ebx
 2: mov    esi,[esp+0xc]   # esi = n
 6: mov    ebx,[esp+0x18]  # ebx = x
 a: vmovsd xmm1,[esp+0x10] # xmm1 = a
 10: mov    ecx,[esp+0x1c]  # ecx = y
 14: vmovddup xmm2,xmm1     # xmm2 = {a,a}
 18: mov    eax,esi
 1a: and    eax,0xfffffffc  # eax = floor(n/4)*4
 1d: vinsertf128 ymm2,ymm2,xmm2,0x1 # ymm2 = {a,a,a,a}
 23: je     3e              # if n < 4 goto Fringe
 25: xor    edx,edx         # edx = 0
Main Loop:
 27: vmovapd ymm0,[ebx+edx*8] # load 4 elements of x
 2c: vfmadd213pd ymm0,ymm2,[ecx+edx*8] # 4 mul adds
 32: vmovapd [ecx+edx*8],ymm0 # store into 4 elements of y
 37: add    edx,0x4
 3a: cmp    edx,eax          # compare to n
 3c: jb     27               # repeat loop if < n
Fringe:
 3e: cmp    esi,eax          # any fringe elements?
 40: jbe    59               # if (n mod 4) == 0 go to Done
Fringe Loop:
 42: vmovsd xmm0,[ebx+eax*8] # load element of x
 47: vfmadd213sd xmm0,xmm1,[ecx+eax*8] # 1 mul add
 4d: vmovsd [ecx+eax*8],xmm0 # store into element of y
 52: add    eax,0x1          # increment Fringe count
 55: cmp    esi,eax          # compare Loop and Fringe counts
 57: jne    42 <daxpy+0x42>  # repeat FringeLoop if != 0
Done:
 59: pop    ebx              # function epilogue
 5a: pop    esi
 5b: ret
```

RISC-V (RVV)

```
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0:  li t0, 2<<25
  4:  vsetdcfg t0            # enable 2 64b Fl.Pt. registers
loop:
  8:  setvl  t0, a0          # vl = t0 = min(mvl, n)
  c:  vld    v0, a1          # load vector x
 10:  slli   t1, t0, 3       # t1 = vl * 8 (in bytes)
 14:  vld    v1, a2          # load vector y
 18:  add    a1, a1, t1      # increment pointer to x by vl*8
 1c:  vfmadd v1, v0, fa0, v1 # v1 += v0 * fa0 (y = a * x + y)
 20:  sub    a0, a0, t0      # n -= vl (t0)
 24:  vst    v1, a2          # store Y
 28:  add    a2, a2, t1      # increment pointer to y by vl*8
 2c:  bnez   a0, loop        # repeat if n != 0
 30:  ret                    # return
```

Example: SIMD vs Vector

```
void daxpy(size_t n, double a, const double x[], double y[])
{
 for (size_t i = 0; i < n; i++) {
   y[i] = a*x[i] + y[i];
 }
}
```

ISA	*MIPS-32 MSA*	*IA-32 AVX2*	*RV32V*
Instructions (static)	22	29	13
Instructions per Main Loop	7	6	10
Bookkeeping Instructions	15	23	3
Results per Main Loop	2	4	64
Instructions (dynamic n=1000)	3511	1517	163

Example: SIMD vs Vector

```
void daxpy(size_t n, double a, const double x[], double y[])
{
 for (size_t i = 0; i < n; i++) {
   y[i] = a*x[i] + y[i];
 }
}
```

ISA	*MIPS-32 MSA*	*IA-32 AVX2*	*RV32V*
Instructions (static)	22	29	13
Instructions per Main Loop	7	6	10
Bookkeeping Instructions	15	23	3
Results per Main Loop	2	4	64
Instructions (dynamic n=1000)	3511	1517	163

This talk...

Let's look at real code

Running on Xilinx Zynq-7020 FPGA (ZedBoard):

1. ARM Cortex-A9 with NEON (667MHz, 128b datapath)
2. ARM Cortex-A9 with RVV (100MHz, 512b datapath)
3. ARM Cortex-A9 with MXP (100MHz, 512b datapath)

Note1: NEON has 1.66x "ops per second" advantage (667MHz/100MHz) * (128b / 512b)

Note2: NEON has 8x more memory bandwidth (6400MB/s vs 800MB/s)

Note3: RISC-V and MXP have 256x more vector data storage (256B vs 64kB)

ARM NEON

16 named registers, 128b wide

128b datapath

4 x 32b operations / cycle

8 x 16b operations / cycle

16 x 8b operations / cycle

supports float32, we'll use int32, int16

Source: ARM NEON Programmers Guide

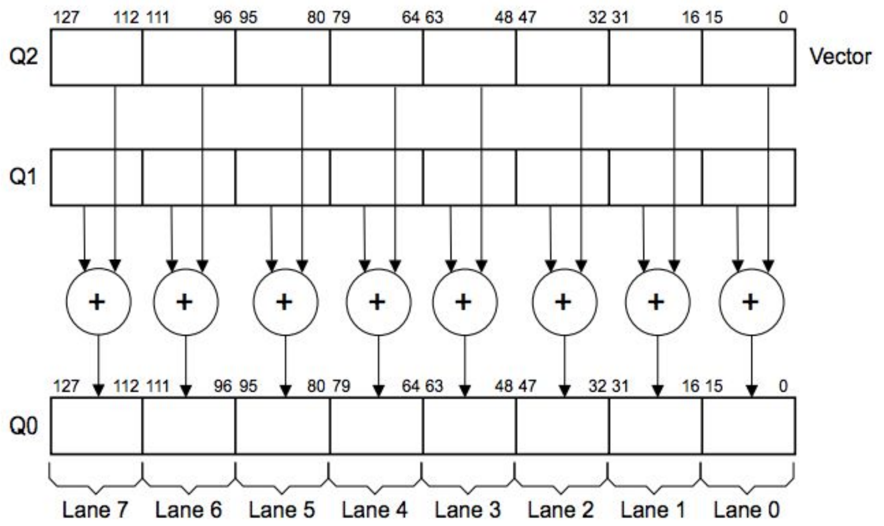


Figure 1-4 8 way 16-bit integer add operation

Note1: wider registers or datapath → ISA must change
Note2: wider registers or datapath → software must be rewritten

RISC-V Vectors

32 named registers

Implementation-defined:

Register size (in bits) ~ MAXVL	16384b (2kB, 512 words)
Execution width (in bits) ~ EXECW	512b
Multiple cycles to execute ~ MAXVL / EXECW	#cycles=32

VectorBlox MXP

No named registers, no MAXVL

All vector data is memory-mapped scratchpad, ie uses scalar pointers

Scratchpad is multi-banked

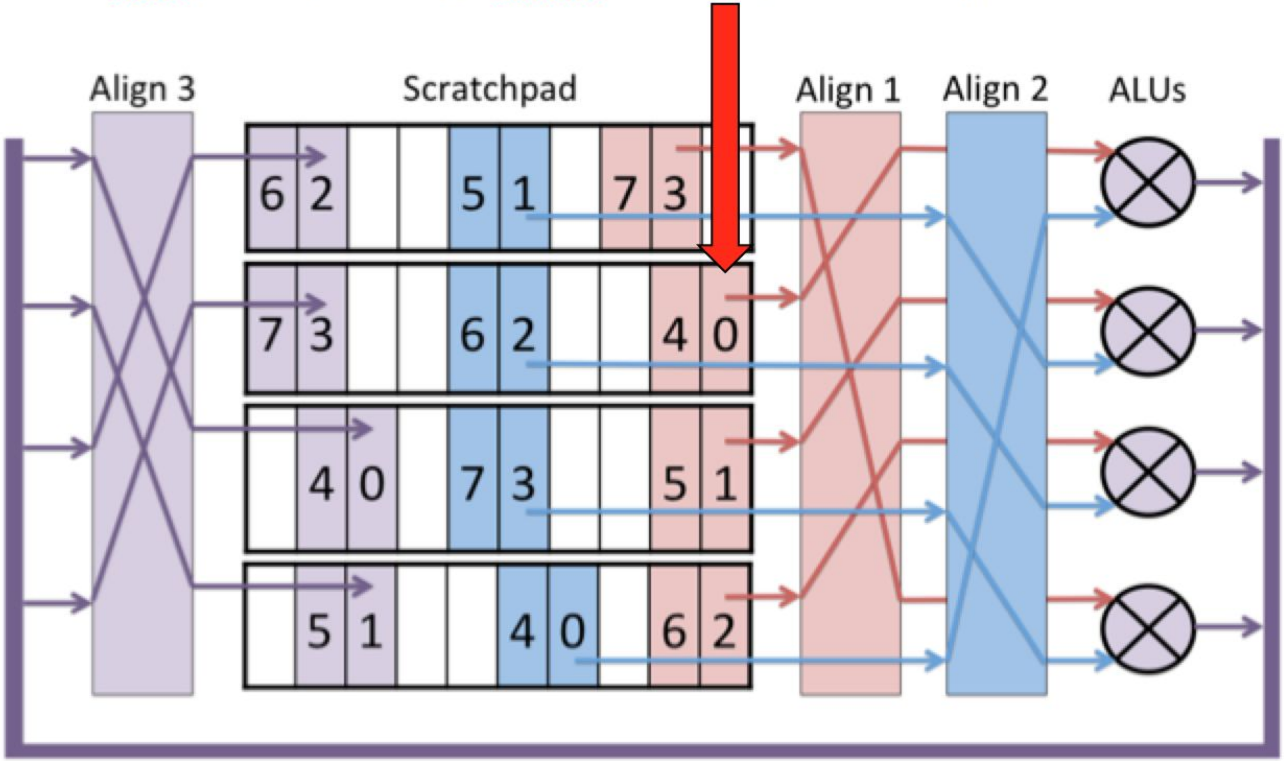
Analogy: file lookup on RAID disk array

Emulate RVV: V0 = base_addr, Vi = V0+i*MAXVL*sizeof(max_elem_size)

Note1: extremely flexible, no vector placement (alignment) or length restrictions
Note2: vector length subject to size of scratchpad

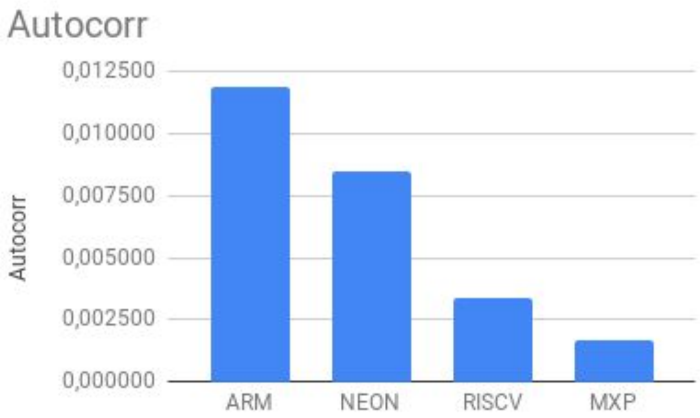
MXP Scratchpad as a RAID disk array

```
vbx_word_t *vdst, *vsrc1, *vsrc2;

vbx( VVW, VADD, vdst, vsrc1, vsrc2 );
```

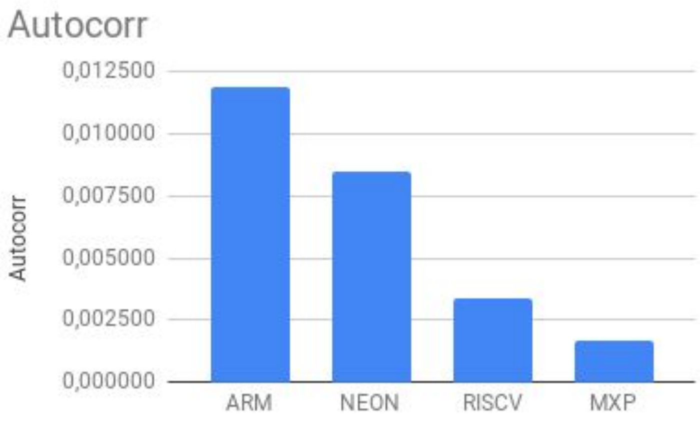


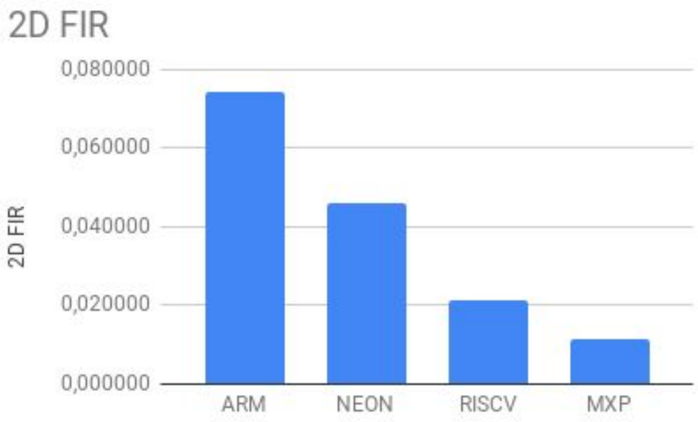
Benchmark Performance

Benchmark Performance

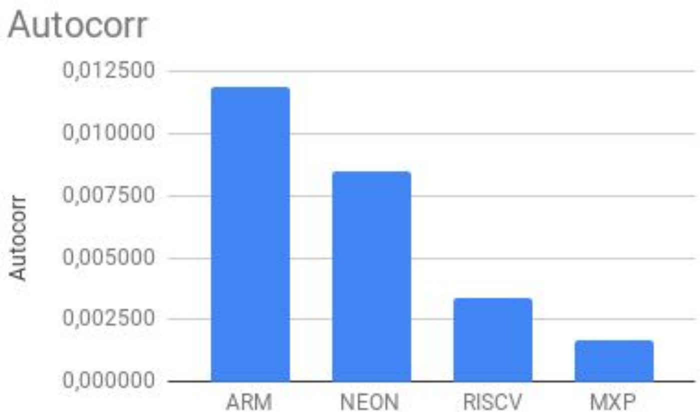


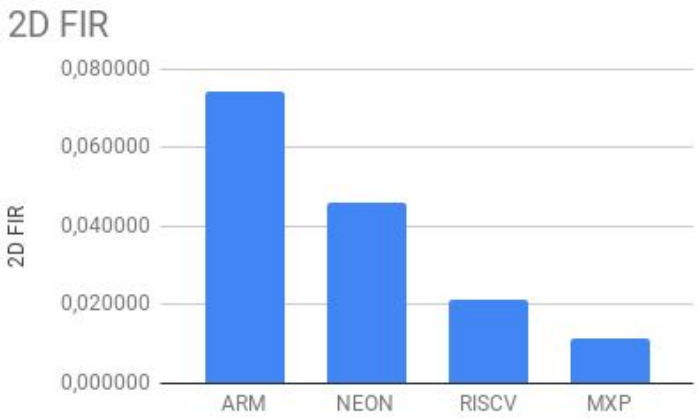
Benchmark Performance

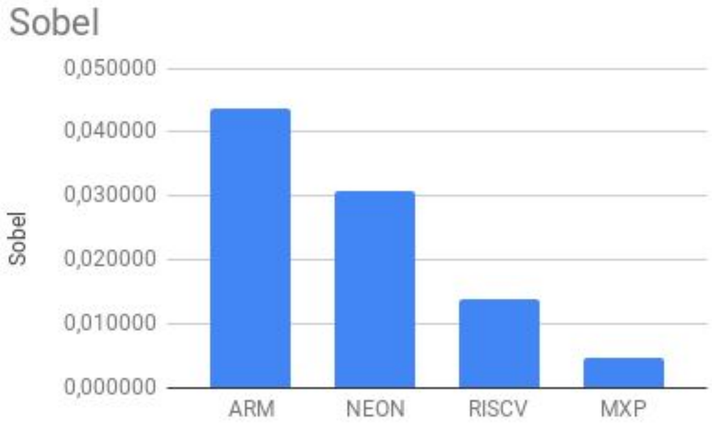




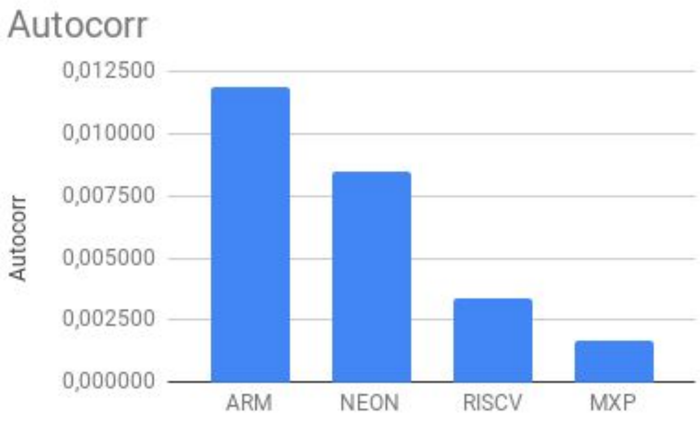
Benchmark Performance

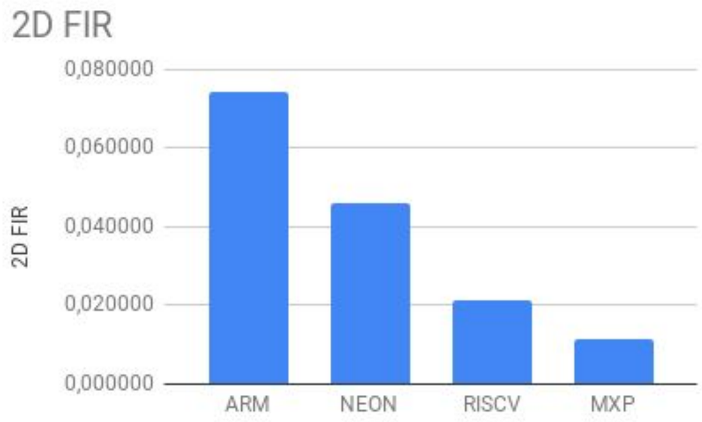


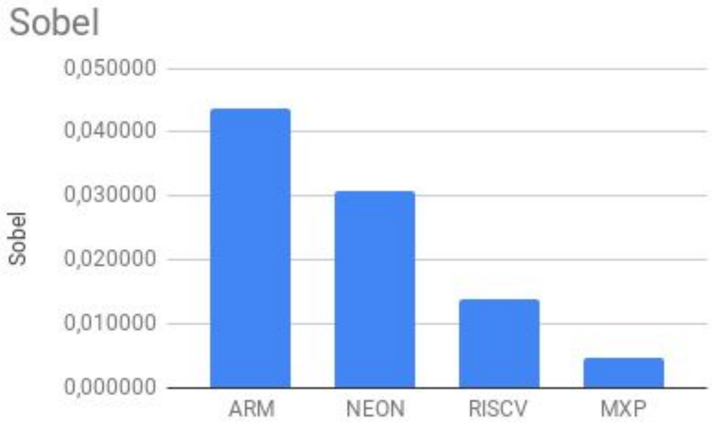


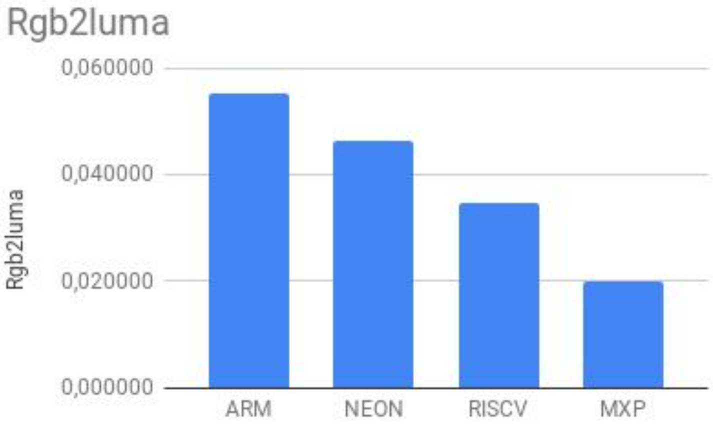


Benchmark Performance



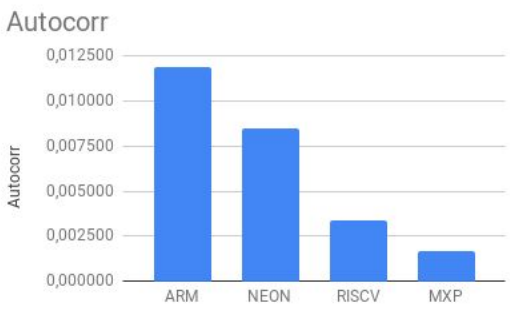






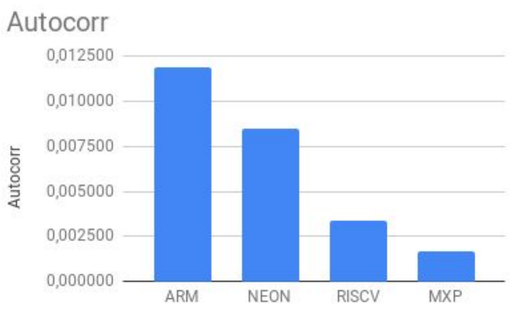
Autocorrelation (32b)

```
for( i=0; i<1800-lag; i++ ) s += ( A[i]*A[i+k] ) >> 2;  // k: 0..1024
```



Autocorrelation (32b)

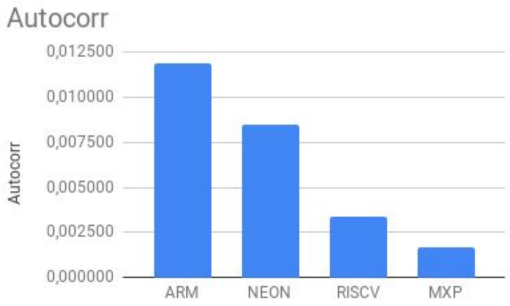
for(i=0; i<1800-lag; i++) s += (A[i]*A[i+k]) >> 2; // k: 0..1024



NEON

```
for(lag = 0; lag < NumberOfLags; lag++){
    uint32x4_t acc_tmp_q = {0, 0, 0, 0};
    for (int i = 0; i < DataSize-lag; i+=4) {
        input     = vld1q_u32(InputData+i);
        input_lag = vld1q_u32(InputData+lag+i);
        mul_q     = vmulq_u32(input,input_lag);
        mul_q     = vshrq_n_u32(mul_q,SCALE_FACTOR);
        acc_tmp_q = vaddq_u32(acc_tmp_q,mul_q);
    }
    // reduce acc_tmp_q, etc ...
}
```

Autocorrelation (32b)



for(i=0; i<1800-lag; i++) s += (A[i]*A[i+k]) >> 2; // k: 0..1024

NEON

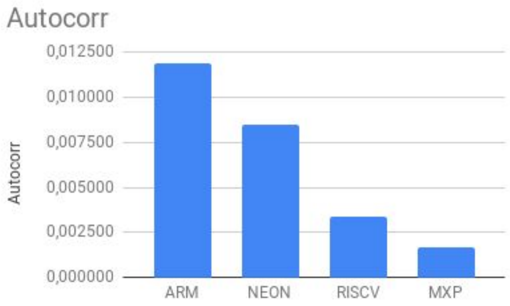
```
for(lag = 0; lag < NumberOfLags; lag++){
    uint32x4_t acc_tmp_q = {0, 0, 0, 0};
    for (int i = 0; i < DataSize-lag; i+=4) {
        input     = vld1q_u32(InputData+i);
        input_lag = vld1q_u32(InputData+lag+i);
        mul_q     = vmulq_u32(input,input_lag);
        mul_q     = vshrq_n_u32(mul_q,SCALE_FACTOR);
        acc_tmp_q = vaddq_u32(acc_tmp_q,mul_q);
    }
    // reduce acc_tmp_q, etc ...
}
```

RISC-V

```
rv_vsetvl(DataSize);
rv_vlwu(v1,InputData);
rv_vlwu(v2,InputData);
for(lag = 0; lag < NumberOfLags; lag++){
    rv_vsetvl( DataSize-lag );
    rv_vslidedn_vs32( v5,v2,lag );
    rv_vmul_vv32( v3,v1,v5 );
    rv_vsrl_vi32( v4,v3,Scale );
    rv_vredsum_v32( v6, v4, v0 );
    rv_vmv_xv32( &sum, v6, 0 );
    rv_vmv_vx32( v7,  sum, lag );
}
```

Autocorrelation (32b)

for(i=0; i<1800-lag; i++) s += (A[i]*A[i+k]) >> 2; // k: 0..1024



NEON

```
for(lag = 0; lag < NumberOfLags; lag++){
    uint32x4_t acc_tmp_q = {0, 0, 0, 0};
    for (int i = 0; i < DataSize-lag; i+=4) {
        input     = vld1q_u32(InputData+i);
        input_lag = vld1q_u32(InputData+lag+i);
        mul_q     = vmulq_u32(input,input_lag);
        mul_q     = vshrq_n_u32(mul_q,SCALE_FACTOR);
        acc_tmp_q = vaddq_u32(acc_tmp_q,mul_q);
    }
    // reduce acc_tmp_q, etc ...
}
```

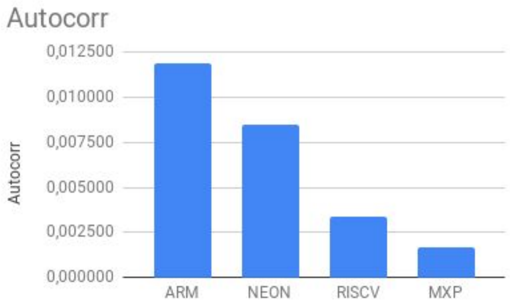
RISC-V

```
rv_vsetvl(DataSize);
rv_vlwu(v1,InputData);
rv_vlwu(v2,InputData);
for(lag = 0; lag < NumberOfLags; lag++){
    rv_vsetvl( DataSize-lag );
    rv_vslidedn_vs32( v5,v2,lag );
    rv_vmul_vv32( v3,v1,v5 );
    rv_vsrl_vi32( v4,v3,Scale );
    rv_vredsum_v32( v6, v4, v0 );
    rv_vmv_xv32( &sum, v6, 0 );
    rv_vmv_vx32( v7,  sum, lag );
}
```

MXP

```
vbx_dma_to_vector( input, InputData, DataSize*4 );
for(lag = 0; lag < NumberOfLags; lag++){
    vbx_set_vl( DataSize-lag,1,1 );
    vbx    ( VVW, VMUL, temp ,input, input+lag );
    vbx_acc( VSW,VSHR,output+lag,temp,Scale );
}
```

Autocorrelation (32b)



for(i=0; i<1800-lag; i++) s += (A[i]*A[i+k]) >> 2; // k: 0..1024

NEON

```
for(lag = 0; lag < NumberOfLags; lag++){
    uint32x4_t acc_tmp_q = {0, 0, 0, 0};
    for (int i = 0; i < DataSize-lag; i+=4) {
        input     = vld1q_u32(InputData+i);
        input_lag = vld1q_u32(InputData+lag+i);
        mul_q     = vmulq_u32(input,input_lag);
        mul_q     = vshrq_n_u32(mul_q,SCALE_FACTOR);
        acc_tmp_q = vaddq_u32(acc_tmp_q,mul_q);
    }
    // reduce acc_tmp_q, etc ...
}
```

RISC-V

```
rv_vsetvl(DataSize);
rv_vlwu(v1,InputData);
rv_vlwu(v2,InputData);
for(lag = 0; lag < NumberOfLags; lag++){
    rv_vsetvl( DataSize-lag );
    rv_vslidedn_vs32( v5,v2,lag );
    rv_vmul_vv32( v3,v1,v5 );
    rv_vsrl_vi32( v4,v3,Scale );
    rv_vredsum_v32( v6, v4, v0 );
    rv_vmv_xv32( &sum, v6, 0 );
    rv_vmv_vx32( v7,  sum, lag );
}
```

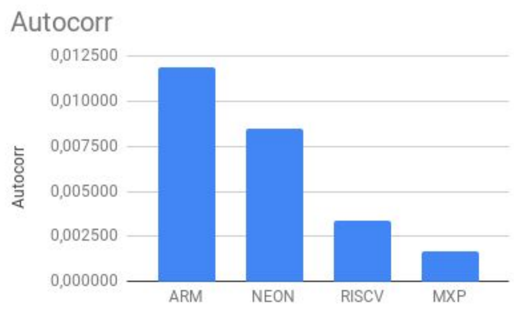
MXP

```
vbx_dma_to_vector( input, InputData, DataSize*4 );
for(lag = 0; lag < NumberOfLags; lag++){
    vbx_set_vl( DataSize-lag,1,1 );
    vbx     ( VVW, VMUL, temp ,input, input+lag );
    vbx_acc( VSW,VSHR,output+lag,temp,Scale );
}
```

Why is MXP faster?

Autocorrelation (32b)

for(i=0; i<1800-lag; i++) s += (A[i]*A[i+k]) >> 2; // k: 0..1024



NEON

```
for(lag = 0; lag < NumberOfLags; lag++){
    uint32x4_t acc_tmp_q = {0, 0, 0, 0};
    for (int i = 0; i < DataSize-lag; i+=4) {
        input     = vld1q_u32(InputData+i);
        input_lag = vld1q_u32(InputData+lag+i);
        mul_q     = vmulq_u32(input,input_lag);
        mul_q     = vshrq_n_u32(mul_q,SCALE_FACTOR);
        acc_tmp_q = vaddq_u32(acc_tmp_q,mul_q);
    }
    // reduce acc_tmp_q, etc ...
}
```

RISC-V

```
rv_vsetvl(DataSize);
rv_vlwu(v1,InputData);
rv_vlwu(v2,InputData);
for(lag = 0; lag < NumberOfLags; lag++){
    rv_vsetvl( DataSize-lag );
    rv_vslidedn_vs32( v5,v2,lag );
    rv_vmul_vv32( v3,v1,v5 );
    rv_vsrl_vi32( v4,v3,Scale );
    rv_vredsum_v32( v6, v4, v0 );
    rv_vmv_xv32( &sum, v6, 0 );
    rv_vmv_vx32( v7,  sum, lag );
}
```

MXP

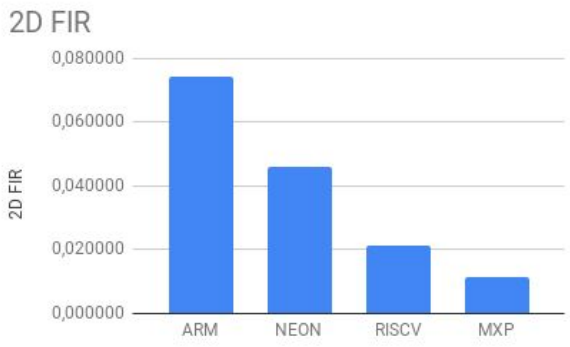
```
vbx_dma_to_vector( input, InputData, DataSize*4 );
for(lag = 0; lag < NumberOfLags; lag++){
    vbx_set_vl( DataSize-lag,1,1 );
    vbx    ( VVW, VMUL, temp ,input, input+lag );
    vbx_acc( VSW,VSHR,output+lag,temp,Scale );
}
```

Why is MXP faster?

RISC-V	MXP
vslidedn (moves data)	scalar increment (start address of vector)
vsrl, vredsum (2 instructions)	(1 instruction) accumulate vshr

2D FIR (32b)

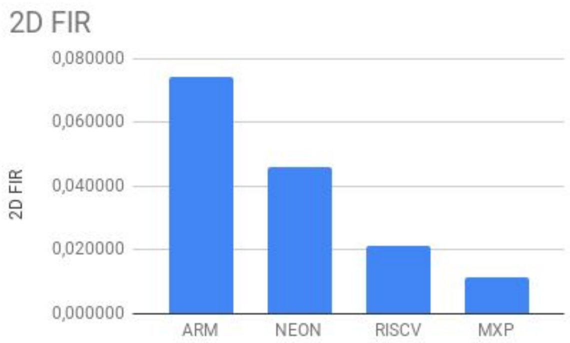
out[x,y] = $\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$ * A[x+dx,y+dy]   // A is 512 x 512



2D FIR (32b)

$$out[x,y] = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} * A[x+dx,y+dy]$$

// A is 512 x 512



RISC-V

```
for( j = 1; j < ntaps_row; j++ ) {
    for( i = 0; i < ntaps_column; i++ ) {
        rv_vslidedn_vs32(v4,v1,modj*num_column+i);
        rv_vmul_vs32(v2,v4,coeffs[j*ntaps_column+i]);
        rv_vadd_vv32(v5,v5,v2);
    }
    modj++;
    if( modj == ntaps_row ) modj = 0;
}
```

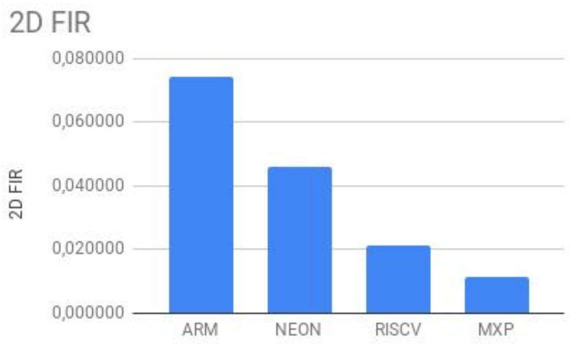
MXP

```
for( j = 1; j < ntaps_row; j++ ) {
  for( i = 0; i < ntaps_column; i++ ) {
    uint32_t      coeff = coeffs[j*ntaps_column+i];
    vbx_uword_t *v_data = sample + modj+num_column+i ;
    vbx(SVWU, VMULLO,  mult_int, coeff, v_data );
    vbx(VVWU, VADD,   accum_int, accum_int, mult_int );
  }
  modj++;
  if( modj == ntaps_row ) modj = 0;
}
```

2D FIR (32b)

$$out[x,y] = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} * A[x+dx,y+dy]$$

// A is 512 x 512



RISC-V

```
for( j = 1; j < ntaps_row; j++ ) {
    for( i = 0; i < ntaps_column; i++ ) {
        rv_vslidedn_vs32(v4,v1,modj*num_column+i);
        rv_vmul_vs32(v2,v4,coeffs[j*ntaps_column+i]);
        rv_vadd_vv32(v5,v5,v2);
    }
    modj++;
    if( modj == ntaps_row ) modj = 0;
}
```

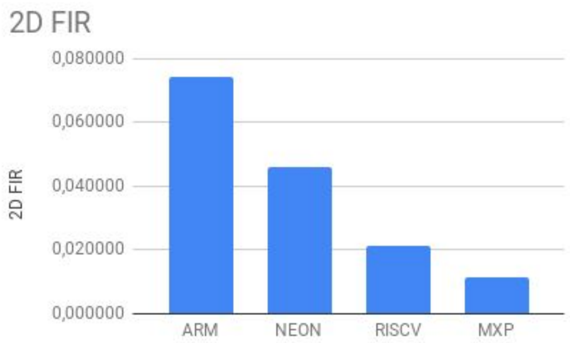
MXP

```
for( j = 1; j < ntaps_row; j++ ) {
  for( i = 0; i < ntaps_column; i++ ) {
    uint32_t      coeff = coeffs[j*ntaps_column+i];
    vbx_uword_t *v_data = sample + modj+num_column+i ;
    vbx(SVWU, VMULLO,  mult_int, coeff, v_data );
    vbx(VVWU, VADD,   accum_int, accum_int, mult_int );
  }
  modj++;
  if( modj == ntaps_row ) modj = 0;
}
```

Why is MXP faster?

2D FIR (32b)

out[x,y] = [a b c d; e f g h; i j k l; m n o p] * A[x+dx,y+dy]   // A is 512 x 512



RISC-V

```
for( j = 1; j < ntaps_row; j++ ) {
    for( i = 0; i < ntaps_column; i++ ) {
        rv_vslidedn_vs32(v4,v1,modj*num_column+i);
        rv_vmul_vs32(v2,v4,coeffs[j*ntaps_column+i]);
        rv_vadd_vv32(v5,v5,v2);
    }
    modj++;
    if( modj == ntaps_row ) modj = 0;
}
```

MXP

```
for( j = 1; j < ntaps_row; j++ ) {
  for( i = 0; i < ntaps_column; i++ ) {
    uint32_t      coeff = coeffs[j*ntaps_column+i];
    vbx_uword_t *v_data = sample + modj+num_column+i ;
    vbx(SVWU, VMULLO,  mult_int, coeff, v_data );
    vbx(VVWU, VADD,   accum_int, accum_int, mult_int );
  }
  modj++;
  if( modj == ntaps_row ) modj = 0;
}
```

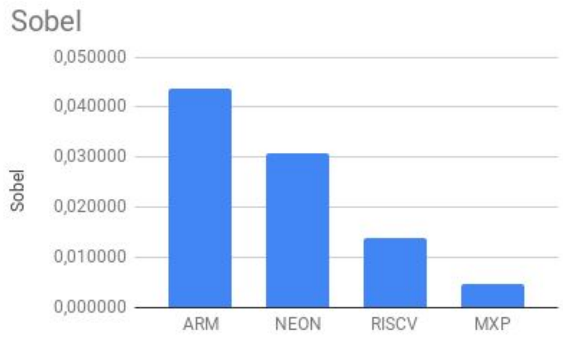
Why is MXP faster?

RISC-V	MXP
vslidedn needed to move data	scalar computes start address of vector

Sobel (16b)

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

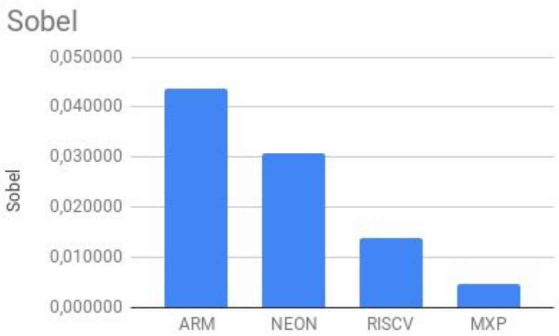
out[x,y] = min(255, ($|G_x|$ + $|G_y|$) >>2);   // input A is 1024 x 1024



Sobel (16b)

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

out[x,y] = min(255, ($|G_x|$ + $|G_y|$) >>2); // input A is 1024 x 1024



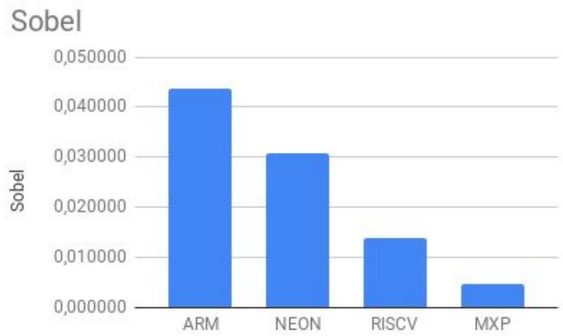
RISC-V

```
for (int i = 0; i < image_height-(FILTER_HEIGHT-1); i++) {
    v_row_in = v_row_in+image_width;
    rv_vsetvl(image_width);
    rv_vlhu(v4,v_row_in);
    rv_vsetvl(image_width-1);
    rv_vslidedn_vi16(v6,v4,1);
    rv_vadd_vv16(v7,v4,v6);
    rv_vsetvl(image_width-2);
    rv_vslidedn_vi16(v8,v7,1);
    rv_vadd_vv16(v10,v7,v8);
    rv_vsetvl(image_width);
    rv_vsll_vi16(v6,v2,1);
    rv_vadd_vv16(v7,v3,v4);
    rv_vadd_vv16(v7,v7,v6);
    rv_vsetvl(image_width-2);
    rv_vslidedn_vi16(v6,v7,2);
    rv_vslt_vv16(v0,v6,v7);         // ABSDIFF
    rv_vsub_vv16(v8,v6,v7);         //
    rv_vsub_vv16(v11,v7,v6);        //
    rv_vmerge_vv16(v8,v11,v8,V0_T); //
    rv_vslt_vv16(v0,v10,v5);        // ABSDIFF
    rv_vsub_vv16(v7,v10,v5);        //
    rv_vsub_vv16(v11,v5,v10);       //
    rv_vmerge_vv16(v7,v11,v7,V0_T); //
    rv_vsetvl(image_width-2);
    rv_vadd_vv16(v6,v7,v8);
    rv_vsrl_vi16(v12,v6,RENORM);
    rv_vsle_vs16(v0,v6,v13);
    rv_vmerge_vs16(v7,v6,v13,V0_F);
    rv_vsb(v7,m_out+(i+1)*image_width+1);
    rv_vsetvl(image_width);
    rv_vor_vv16( v11, v3 , v3  ); // rotate buffers
    rv_vor_vv16( v3 , v2 , v2  );
    // etc
}
```

MXP

```
for (y = 0; y < image_height-(FILTER_HEIGHT-1); y++) {
    v_tmp = v_sobel_row_bot;
    vbx_set_vl(image_width,1,1);
    vbx(VSHU, VSHL, v_gradient_x, v_luma_mid, 1);
    vbx(VVHU, VADD, v_tmp,        v_luma_top, v_luma_bot);
    vbx(VVHU, VADD, v_tmp,        v_tmp,      v_gradient_x);
    luma_input = luma_input + image_width;
    vbx_dma_to_vector(v_luma_top, luma_input, image_width*sizeof(vbx_uhalf_t));
    vbx_set_vl(image_width-1,1,1);
    vbx(VVHU, VADD, v_sobel_row_bot, v_luma_bot, v_luma_bot+1);
    vbx_set_vl(image_width-2,1,1);
    vbx(VVHU, VADD, v_sobel_row_bot, v_sobel_row_bot, v_sobel_row_bot+1);
    vbx(VVH, VABSDIFF, v_gradient_x, v_tmp, v_tmp+2);
    vbx(VVH, VABSDIFF, v_gradient_y, v_sobel_row_top, v_sobel_row_bot);
    v_tmp = v_sobel_row_top;
    vbx_set_vl(image_width-2,1,1);
    vbx(VVHU, VADD, v_tmp, v_gradient_x,  v_gradient_y);
    vbx(VSHU, VSHR, v_tmp, v_tmp, renorm);
    vbx(SVHU,   VSUB,     v_gradient_y, 255, v_tmp);
    vbx(SVBHHU, VCMV_LTZ, v_row_out+1, 255, v_gradient_y);
    vbx_dma_to_host(output+(y+1)*image_pitch,
            v_row_out, (image_width)*sizeof(vbx_ubyte_t));
    tmp_ptr      = (void *)v_luma_top; // rotate buffers
    v_luma_top   = v_luma_mid;
    // etc
}
```

Sobel (16b)



$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

out[x,y] = min(255, ($|G_x|$ + $|G_y|$) >>2);   // input A is 1024 x 1024

RISC-V

Why is MXP faster?

```
for (int i = 0; i < image_height-(FILTER_HEIGHT-1); i++) {
    v_row_in = v_row_in+image_width;
    rv_vsetvl(image_width);
    rv_vlhu(v4,v_row_in);
    rv_vsetvl(image_width-1);
    rv_vslidedn_vi16(v6,v4,1);
    rv_vadd_vv16(v7,v4,v6);
    rv_vsetvl(image_width-2);
    rv_vslidedn_vi16(v8,v7,1);
    rv_vadd_vv16(v10,v7,v8);
    rv_vsetvl(image_width);
    rv_vsll_vi16(v6,v2,1);
    rv_vadd_vv16(v7,v3,v4);
    rv_vadd_vv16(v7,v7,v6);
    rv_vsetvl(image_width-2);
    rv_vslidedn_vi16(v6,v7,2);
    rv_vslt_vv16(v0,v6,v7);         // ABSDIFF
    rv_vsub_vv16(v8,v6,v7);         //
    rv_vsub_vv16(v11,v7,v6);        //
    rv_vmerge_vv16(v8,v11,v8,V0_T); //
    rv_vslt_vv16(v0,v10,v5);        // ABSDIFF
    rv_vsub_vv16(v7,v10,v5);        //
    rv_vsub_vv16(v11,v5,v10);       //
    rv_vmerge_vv16(v7,v11,v7,V0_T); //
    rv_vsetvl(image_width-2);
    rv_vadd_vv16(v6,v7,v8);
    rv_vsrl_vi16(v12,v6,RENORM);
    rv_vsle_vs16(v0,v6,v13);
    rv_vmerge_vs16(v7,v6,v13,V0_F);
    rv_vsb(v7,m_out+(i+1)*image_width+1);
    rv_vsetvl(image_width);
    rv_vor_vv16( v11, v3 , v3  ); // rotate buffers
    rv_vor_vv16( v3 , v2 , v2  );
    // etc
}
```

RISC-V	MXP
20 arith operations	11 total arith operations
3 vslidedn	0 slides (scalar add)
4 instructions per absdiff	2 absdiff instructions
6 data move operations	0 data move (scalar ptr assign.)

MXP

```
for (y = 0; y < image_height-(FILTER_HEIGHT-1); y++) {
    v_tmp = v_sobel_row_bot;
    vbx_set_vl(image_width,1,1);
    vbx(VSHU, VSHL, v_gradient_x, v_luma_mid, 1);
    vbx(VVHU, VADD, v_tmp,        v_luma_top, v_luma_bot);
    vbx(VVHU, VADD, v_tmp,        v_tmp,      v_gradient_x);
    luma_input = luma_input + image_width;
    vbx_dma_to_vector(v_luma_top, luma_input, image_width*sizeof(vbx_uhalf_t));
    vbx_set_vl(image_width-1,1,1);
    vbx(VVHU, VADD, v_sobel_row_bot, v_luma_bot, v_luma_bot+1);
    vbx_set_vl(image_width-2,1,1);
    vbx(VVHU, VADD, v_sobel_row_bot, v_sobel_row_bot, v_sobel_row_bot+1);
    vbx(VVH, VABSDIFF, v_gradient_x, v_tmp, v_tmp+2);
    vbx(VVH, VABSDIFF, v_gradient_y, v_sobel_row_top, v_sobel_row_bot);
    v_tmp = v_sobel_row_top;
    vbx_set_vl(image_width-2,1,1);
    vbx(VVHU, VADD, v_tmp, v_gradient_x,  v_gradient_y);
    vbx(VSHU, VSHR, v_tmp, v_tmp, renorm);
    vbx(SVHU,   VSUB,     v_gradient_y, 255, v_tmp);
    vbx(SVBHHU, VCMV_LTZ, v_row_out+1, 255, v_gradient_y);
    vbx_dma_to_host(output+(y+1)*image_pitch,
            v_row_out, (image_width)*sizeof(vbx_ubyte_t));
    tmp_ptr      = (void *)v_luma_top; // rotate buffers
    v_luma_top   = v_luma_mid;
    // etc
}
```

RISC-V

```
for (int i = 0; i < image_height-(FILTER_HEIGHT-1); i++) {
    v_row_in = v_row_in+image_width;
    rv_vsetvl(image_width);
    rv_vlhu(v4,v_row_in);
    rv_vsetvl(image_width-1);
    rv_vslidedn_vi16(v6,v4,1);
    rv_vadd_vv16(v7,v4,v6);
    rv_vsetvl(image_width-2);
    rv_vslidedn_vi16(v8,v7,1);
    rv_vadd_vv16(v10,v7,v8);
    rv_vsetvl(image_width);
    rv_vsll_vi16(v6,v2,1);
    rv_vadd_vv16(v7,v3,v4);
    rv_vadd_vv16(v7,v7,v6);
    rv_vsetvl(image_width-2);
    rv_vslidedn_vi16(v6,v7,2);
    rv_vslt_vv16(v0,v6,v7);          // ABSDIFF
    rv_vsub_vv16(v8,v6,v7);          //
    rv_vsub_vv16(v11,v7,v6);         //
    rv_vmerge_vv16(v8,v11,v8,V0_T); //
    rv_vslt_vv16(v0,v10,v5);         // ABSDIFF
    rv_vsub_vv16(v7,v10,v5);         //
    rv_vsub_vv16(v11,v5,v10);        //
    rv_vmerge_vv16(v7,v11,v7,V0_T); //
    rv_vsetvl(image_width-2);
    rv_vadd_vv16(v6,v7,v8);
    rv_vsrl_vi16(v12,v6,RENORM);
    rv_vsle_vs16(v0,v6,v13);
    rv_vmerge_vs16(v7,v6,v13,V0_F);
    rv_vsb(v7,m_out+(i+1)*image_width+1);
    rv_vsetvl(image_width);
    rv_vor_vv16( v11, v3 , v3  ); // rotate buffers
    rv_vor_vv16( v3 , v2 , v2  );
    // etc
}
```

MXP

```
for (y = 0; y < image_height-(FILTER_HEIGHT-1); y++) {
    v_tmp = v_sobel_row_bot;
    vbx_set_vl(image_width,1,1);
    vbx(VSHU, VSHL, v_gradient_x, v_luma_mid, 1);
    vbx(VVHU, VADD, v_tmp,        v_luma_top, v_luma_bot);
    vbx(VVHU, VADD, v_tmp,        v_tmp,      v_gradient_x);
    luma_input = luma_input + image_width;
    vbx_dma_to_vector(v_luma_top, luma_input, image_width*sizeof(vbx_uhalf_
    vbx_set_vl(image_width-1,1,1);
    vbx(VVHU, VADD, v_sobel_row_bot, v_luma_bot, v_luma_bot+1);
    vbx_set_vl(image_width-2,1,1);
    vbx(VVHU, VADD, v_sobel_row_bot, v_sobel_row_bot, v_sobel_row_bot+1);
    vbx(VVH, VABSDIFF, v_gradient_x, v_tmp, v_tmp+2);
    vbx(VVH, VABSDIFF, v_gradient_y, v_sobel_row_top, v_sobel_row_bot);
    v_tmp = v_sobel_row_top;
    vbx_set_vl(image_width-2,1,1);
    vbx(VVHU, VADD, v_tmp, v_gradient_x,  v_gradient_y);
    vbx(VSHU, VSHR, v_tmp, v_tmp, renorm);
    vbx(SVHU,   VSUB,     v_gradient_y, 255, v_tmp);
    vbx(SVBHHU, VCMV_LTZ, v_row_out+1, 255, v_gradient_y);
    vbx_dma_to_host(output+(y+1)*image_pitch,
            v_row_out, (image_width)*sizeof(vbx_ubyte_t));
    tmp_ptr     = (void *)v_luma_top; // rotate buffers
    v_luma_top  = v_luma_mid;
    // etc
}
```

Why is MXP faster?

20 vs 11 arith operations

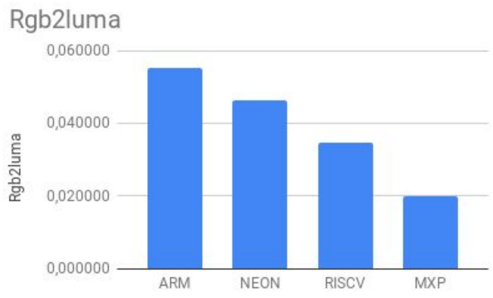
- 3 vslidedn
- 4 instructions per absdiff

6 vs 0 data move operations

RGBA to LUMA (32b → 16b → 8b)

luma8 = uint16(25*blu8 + 129*grn8 + 66*red8 + 128) >> 8

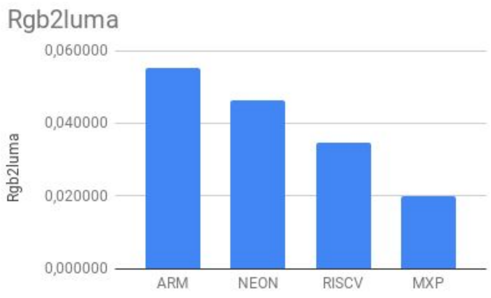
// input is 1600 x 1600



RGBA to LUMA (32b → 16b → 8b)

luma8 = uint16(25*blu8 + 129*grn8 + 66*red8 + 128) >> 8

// input is 1600 x 1600



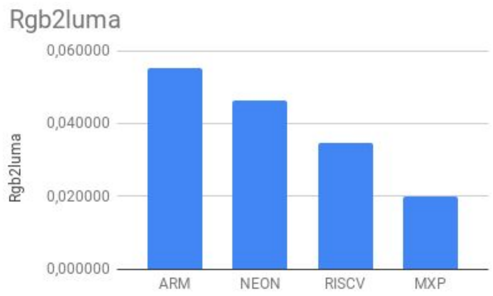
NEON

```
for (int i = 0; i < image_width; i+=4) {
    v_rowin_q       = vld1q_u32(&B_ptr[i]);
    v_rowtemp_q     = vandq_u32(v_rowin_q,const_255);
    v_luma_q        = vmulq_n_u32(v_rowtemp_q,25);

    v_rowin_q       = vld1q_u32(&G_ptr[i]);
    v_rowtemp_q     = vandq_u32(v_rowin_q,const_255);
    v_rowtemp_q     = vmulq_n_u32(v_rowtemp_q,129);
    v_luma_q        = vaddq_u32(v_luma_q,v_rowtemp_q);

    v_rowin_q       = vld1q_u32(&R_ptr[i]);
    v_rowtemp_q     = vandq_u32(v_rowin_q,const_255);
    v_rowtemp_q     = vmulq_n_u32(v_rowtemp_q,66);
    v_luma_q        = vaddq_u32(v_luma_q,v_rowtemp_q);

    v_luma_q        = vaddq_u32(v_luma_q,const_128);
    v_luma_q        = vshrq_n_u32(v_luma_q,8);
    conv_narrow = vmovn_u32(v_luma_q);
    vst1_u16(m_out+(j*image_width)+i,conv_narrow);
}
```

RGBA to LUMA (32b → 16b → 8b)

luma8 = uint16(25*blu8 + 129*grn8 + 66*red8 + 128) >> 8

// input is 1600 x 1600



NEON

```
for (int i = 0; i < image_width; i+=4) {
    v_rowin_q       = vld1q_u32(&B_ptr[i]);
    v_rowtemp_q     = vandq_u32(v_rowin_q,const_255);
    v_luma_q        = vmulq_n_u32(v_rowtemp_q,25);

    v_rowin_q       = vld1q_u32(&G_ptr[i]);
    v_rowtemp_q     = vandq_u32(v_rowin_q,const_255);
    v_rowtemp_q     = vmulq_n_u32(v_rowtemp_q,129);
    v_luma_q        = vaddq_u32(v_luma_q,v_rowtemp_q);

    v_rowin_q       = vld1q_u32(&R_ptr[i]);
    v_rowtemp_q     = vandq_u32(v_rowin_q,const_255);
    v_rowtemp_q     = vmulq_n_u32(v_rowtemp_q,66);
    v_luma_q        = vaddq_u32(v_luma_q,v_rowtemp_q);

    v_luma_q        = vaddq_u32(v_luma_q,const_128);
    v_luma_q        = vshrq_n_u32(v_luma_q,8);
    conv_narrow = vmovn_u32(v_luma_q);
    vst1_u16(m_out+(j*image_width)+i,conv_narrow);
}
```

RISC-V

```
rv_vlwu( v6, v_row_in );

rv_vsrln_wi32( v7, v6, 0 );
rv_vand_vv16( v7, v7, v1 );
rv_vmul_vv16( v8, v2, v7 );

rv_vsrln_wi32( v7, v6, 8 );
rv_vand_vv16( v7, v7, v1 );
rv_vmul_vv16( v9, v3, v7 );
rv_vadd_vv16( v8, v8, v9 );

rv_vsrln_wi32( v7, v6, 16 );
rv_vand_vv16( v7, v7, v1 );
rv_vmul_vv16( v9, v3, v7 );
rv_vadd_vv16( v8, v8, v9 );

rv_vadd_vv16( v8, v8, v5 );
rv_vsrl_vi16( v8, v8, 8 );

rv_vsh( v8, m_out+i*image_width );
```

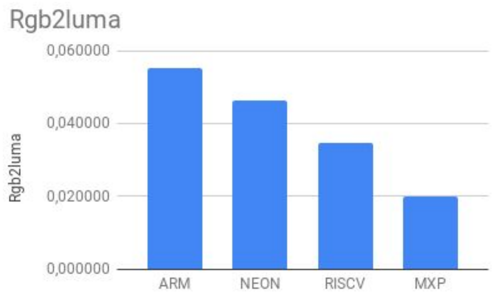
MXP

```
// Move weighted B into v_luma
vbx(SVHWU, VAND, v_temp, 0xFF,    v_row_in);
vbx(SVHU,  VMUL, v_luma, 25,      v_temp);

// Move weighted G into v_temp and add it to v_luma
vbx(SVHWU, VAND, v_temp, 0xFF,    (vbx_uword_t*)(((vbx_ubyte_t *)v_row_in)+1));
vbx(SVHU,  VMUL, v_temp, 129,     v_temp);
vbx(VVHU,  VADD, v_luma, v_luma, v_temp);

// Move weighted R into v_temp and add it to v_luma
vbx(SVHWU, VAND, v_temp, 0xFF,    (vbx_uword_t*)(((vbx_ubyte_t *)v_row_in)+2));
vbx(SVHU,  VMUL, v_temp, 66,      v_temp);
vbx(VVHU,  VADD, v_luma, v_luma, v_temp);

vbx(SVHU,  VADD, v_luma, 128,     v_luma); // for rounding
vbx(VSHU,  VSHR, v_luma, v_luma, 8);

vbx_dma_to_host(m_out+i*image_pitch, v_luma,
        image_pitch*sizeof(vbx_uhalf_t));

rp_fetch(&v_row_db);
v_row_in=rp_get_buffer(&v_row_db,0);
```

RGBA to LUMA (32b → 16b → 8b)

luma8 = uint16(25*blu8 + 129*grn8 + 66*red8 + 128) >> 8

// input is 1600 x 1600



NEON

```
for (int i = 0; i < image_width; i+=4) {
    v_rowin_q      = vld1q_u32(&B_ptr[i]);
    v_rowtemp_q    = vandq_u32(v_rowin_q,const_255);
    v_luma_q       = vmulq_n_u32(v_rowtemp_q,25);

    v_rowin_q      = vld1q_u32(&G_ptr[i]);
    v_rowtemp_q    = vandq_u32(v_rowin_q,const_255);
    v_rowtemp_q    = vmulq_n_u32(v_rowtemp_q,129);
    v_luma_q       = vaddq_u32(v_luma_q,v_rowtemp_q);

    v_rowin_q      = vld1q_u32(&R_ptr[i]);
    v_rowtemp_q    = vandq_u32(v_rowin_q,const_255);
    v_rowtemp_q    = vmulq_n_u32(v_rowtemp_q,66);
    v_luma_q       = vaddq_u32(v_luma_q,v_rowtemp_q);

    v_luma_q       = vaddq_u32(v_luma_q,const_128);
    v_luma_q       = vshrq_n_u32(v_luma_q,8);
    conv_narrow = vmovn_u32(v_luma_q);
    vst1_u16(m_out+(j*image_width)+i,conv_narrow);
}
```

RISC-V

```
rv_vlwu( v6, v_row_in );

rv_vsrln_wi32( v7, v6, 0 );
rv_vand_vv16( v7, v7, v1 );
rv_vmul_vv16( v8, v2, v7 );

rv_vsrln_wi32( v7, v6, 8 );
rv_vand_vv16( v7, v7, v1 );
rv_vmul_vv16( v9, v3, v7 );
rv_vadd_vv16( v8, v8, v9 );

rv_vsrln_wi32( v7, v6, 16 );
rv_vand_vv16( v7, v7, v1 );
rv_vmul_vv16( v9, v3, v7 );
rv_vadd_vv16( v8, v8, v9 );

rv_vadd_vv16( v8, v8, v5 );
rv_vsrl_vi16( v8, v8, 8 );

rv_vsh( v8, m_out+i*image_width );
```

MXP

```
// Move weighted B into v_luma
vbx(SVHWU, VAND, v_temp, 0xFF,    v_row_in);
vbx(SVHU,  VMUL, v_luma, 25,      v_temp);

// Move weighted G into v_temp and add it to v_luma
vbx(SVHWU, VAND, v_temp, 0xFF,    (vbx_uword_t*)(((vbx_ubyte_t *)v_row_in)+1));
vbx(SVHU,  VMUL, v_temp, 129,     v_temp);
vbx(VVHU,  VADD, v_luma, v_luma, v_temp);

// Move weighted R into v_temp and add it to v_luma
vbx(SVHWU, VAND, v_temp, 0xFF,    (vbx_uword_t*)(((vbx_ubyte_t *)v_row_in)+2));
vbx(SVHU,  VMUL, v_temp, 66,      v_temp);
vbx(VVHU,  VADD, v_luma, v_luma, v_temp);

vbx(SVHU,  VADD, v_luma, 128,     v_luma); // for rounding
vbx(VSHU,  VSHR, v_luma, v_luma, 8);

vbx_dma_to_host(m_out+i*image_pitch, v_luma,
        image_pitch*sizeof(vbx_uhalf_t));

rp_fetch(&v_row_db);
v_row_in=rp_get_buffer(&v_row_db,0);
```

Why is MXP faster?

RISC-V	MXP
Narrows data with 3 vsrln instructions	Narrows data with any instruction (vand)
Vector load not prefetched	Row data is prefetched

Summary

RVV vs NEON

already strong performance advantage

RVV vs MXP

1. Extra data movement
   a. vslidedn/up take extra time (eg, sliding windows)
   b. rotating buffers require vmov
2. Unbundled operations
   a. Reductions take an extra instruction
   b. Data-element narrowing takes an extra instruction
3. Missing operations
   a. vabsdiff takes 4 instructions (vslt, vsub, vsub, vmerge)