CEVA®
The DSP Powerhouse

CEVA-XM4™
Coding On CEVA-DSP

2016

www.ceva-dsp.com

Agenda

- Glossary
- Median Filter 3X3
  - Algorithm Analysis
  - Efficient VEC-C Code
  - Performance
- Gaussian3X3
  - Gaussian3X3 Kernel Overview
  - Gaussian3X3 Vector Operators Code
  - Gaussian3X3 load Operation Optimizations
  - Gaussian3X3 Sliding window code

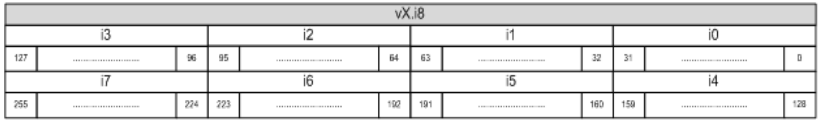
Agenda

- Glossary
- Median Filter 3X3
  - Algorithm Analysis
  - Efficient VEC-C Code
  - Performance
- Gaussian3X3
  - Gaussian3X3 Kernel Overview
  - Gaussian3X3 Vector Operators Code
  - Gaussian3X3 load Operation Optimizations
  - Gaussian3X3 Sliding window code

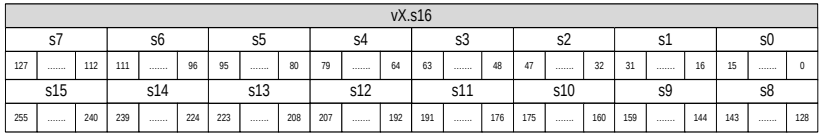
Glossary

- Vector types – newly defined CEVA vector native types to be used in vector operation and mapped to the VPU vectors resources

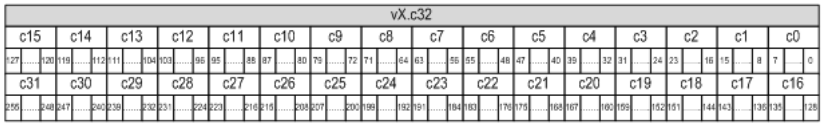
  - **8 integer vector**



  - **16 short vector**



  - **32 byte vector**



Native 256-bit vectors	
char32	uchar32
short16	ushort16
int8	uint8
128-bit vectors	
char16	uchar16
short8	ushort8
512-bit vectors	
short32	ushort32
int16	uint16

Glossary

- Vector operators –
  Simple way of coding an optimized Vec-C code, using vector operators:

```
uchar32 v00, v01, v02;
uchar* p_in_u8 = (uchar *)&p_u8Src[i * stride];

v00 = *(uchar32*)p_in_u8;
p_in_u8++;
v01 = *(uchar32*)p_in_u8;
p_in_u8++;
vacc = (ushort32)v00 * 2 + (ushort32)v01 * 4;
```

- The code is written in a plain way
- Vector operators are used with the vector type as done for other native C language types [char, short or int].
- Ensures fast and simple code development
- Does not give access to all instruction set features

Glossary

- Vec-C – vector intrinsic that have one to one mapping with the instruction set:

```
uchar32 v00, v01, v02;
uchar* p_in_u8 = (uchar *)&p_u8Src[i * 32];
uint16 vacc0;
v00 = *(uchar32*)p_in_u8;
p_in_u8+=s32SrcStep;
v10 = *(uchar32*)p_in_u8;
p_in_u8+=s32SrcStep;
vacc0 = (uint16)vswmpy5(v00, v00, v_coeff, config0);
```

- Enables to use CEVA-XM4 rich instruction set features from a C level environment.
- Provides superb performance per invested coding time.

Agenda

- Glossary
- **Median Filter 3X3**
  - Algorithm Analysis
  - Efficient VEC-C Code
  - Performance
- Gaussian3X3
  - Gaussian3X3 Kernel Overview
  - Gaussian3X3 Vector Operators Code
  - Gaussian3X3 load Operation Optimizations
  - Gaussian3X3 Sliding window code

Median3X3

- **Algorithm Analysis**
- **Efficient VEC-C Code**
- **Performance**

Median3*3 Introduction

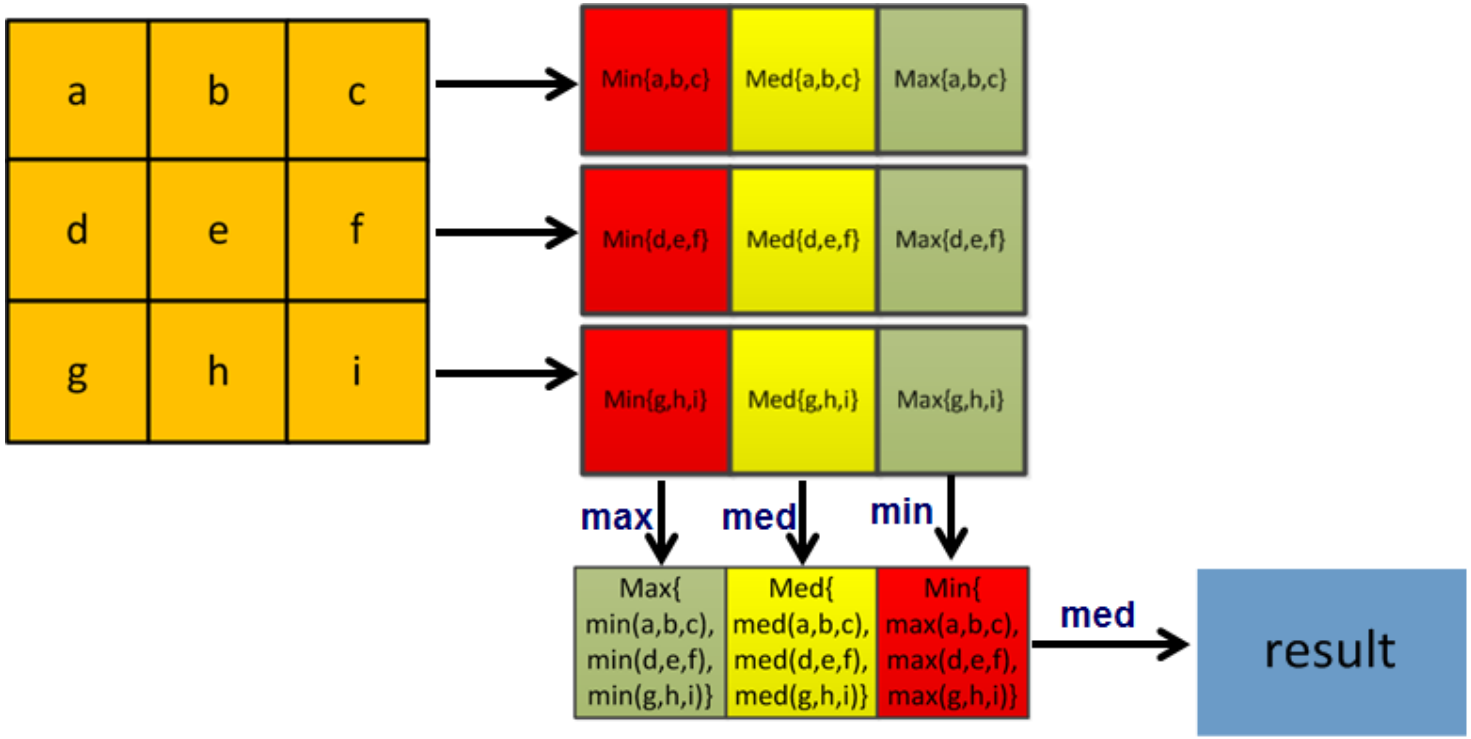
- Dst[x,y] = Median{

  Src[x-1,y-1], Src[x,y-1], Src[x+1,y-1]

  Src[x-1,y] , Src[x,y] , Src[x+1,y]

  Src[x-1,y+1], Src[x,y+1],Src[x+1,y+1]}

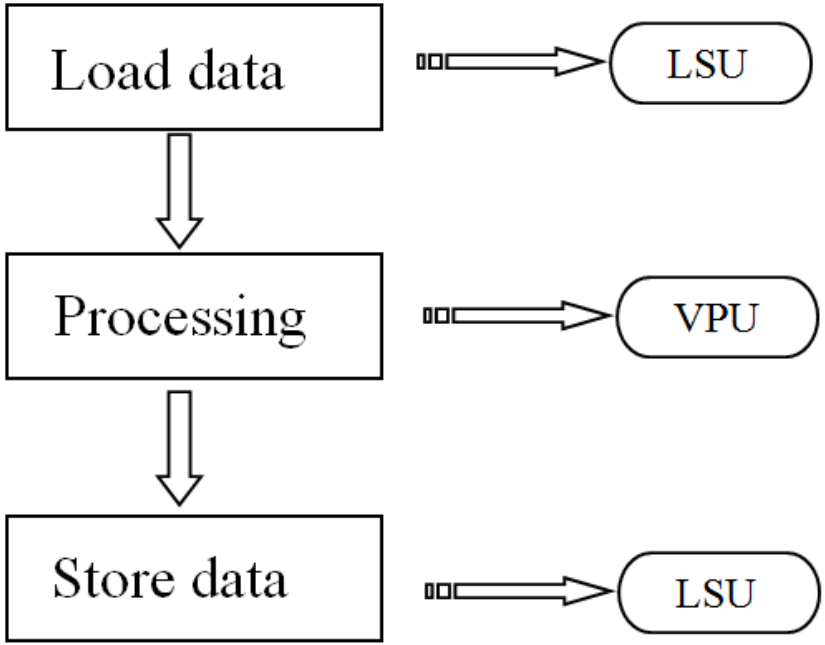
Implementation



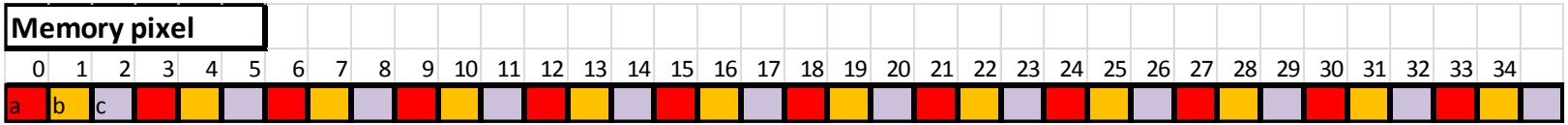
Median3X3

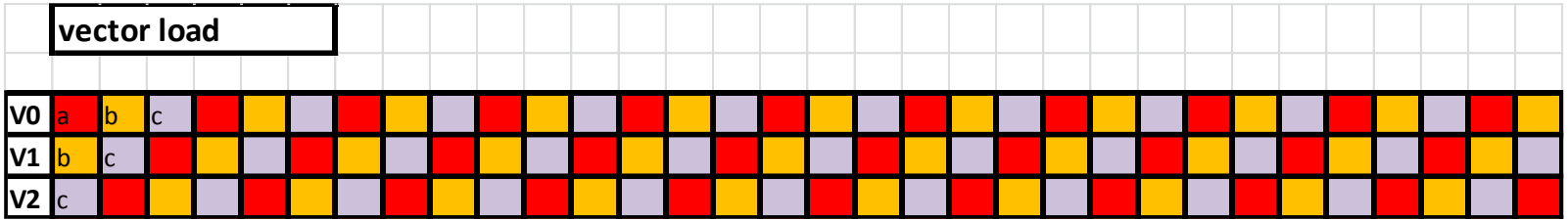
- Algorithm Analysis
- **Efficient VEC-C Code**
- Performance Analysis

Data Process Flow



Loading Data






```
vldov((uchar32*)p_in_u8, v0, v1, v2, vdummy);
```

Processing and Store

```
v0min = vmin(v1, v2, v3);
v0med = vmed(v1, v2, v3);
v0max = vmax(v1, v2, v3);
v1min = vmin(v4, v5, v6);
v1med = vmed(v4, v5, v6);
v1max = vmax(v4, v5, v6);
v2min = vmin(v7, v8, v9);
v2med = vmed(v7, v8, v9);
v2max = vmax(v7, v8, v9);
v1 = vmax(v0min, v1min, v2min);
v2 = vmed(v0med, v1med, v2med);
v3 = vmin(v0max, v1max, v2max);
uchar32 result = vmed(v1, v2, v3);
vst(result, p_dst, vprMask2);
```

Performance

	median3x3
load instruction	vldov(p_src, v1, v2, v3, vdummy)
vpu instruction	vmax(v1, v2, v3)
	vmed(v1, v2, v3)
	vmin(v1, v2, v3)
store instruction	vst(result, p_dst, vprMask)

Agenda

- Glossary
- Median Filter 3X3
  - Algorithm Analysis
  - Efficient VEC-C Code
  - Performance
- **Gaussian3X3**
  - Gaussian3X3 Kernel Overview
  - Gaussian3X3 Vector Operators Code
  - Gaussian3X3 load Operation Optimizations
  - Gaussian3X3 Sliding window code

Gaussian3X3 Kernel Overview

- Gaussian3x3 coefficients {2,4,2,4,8,4,2,4,2}
- Res = $\frac{1}{32}\sum_{i=0}^{i<3}\sum_{j=0}^{j<3} Source_{pixel(Xpos+i,Ypos+j)} * Guas\_coef(i,j)$
- C reference code to be optimized

2	4	2
4	8	4
2	4	2

```
for (i = 0; i <u32M; i++)
{
        uint acc = 0;
        acc +=  p_in_u8[-1 - s32SrcStirde] * s8Kernel[0];
        acc +=  p_in_u8[0 - s32SrcStirde] * s8Kernel[1];
        acc +=  p_in_u8[1 - s32SrcStirde] * s8Kernel[2];
        acc +=  p_in_u8[-1            ] * s8Kernel[3];
        acc +=  p_in_u8[ 0            ] * s8Kernel[4];
        acc +=  p_in_u8[ 1            ] * s8Kernel[5];
        acc +=  p_in_u8[-1 + s32SrcStirde] * s8Kernel[6];
        acc +=  p_in_u8[0 + s32SrcStirde] * s8Kernel[7];
        acc +=  p_in_u8[1 + s32SrcStirde] * s8Kernel[8];
        res = (acc >> ps) &0xFFFF;
        sat = (res < 0 ) ?  0  : (res>255) ? 255 : (uchar)res;
        p_out_u8[i] = sat;
        p_in_u8++;
}
```

Gaussian3X3 Vector Operators Code

- Check that CEVA_OPPERATORS is open in the gaussian3x3_lib.h file.
- Check that the following defines are closed:
  - CEVA_VLDOV and CEVA_SLIDING_WINDOW
- Algorithm flow:
  - Load operation :

```
v00 = *(uchar32*)p_in_u8;
p_in_u8++;
```

  - Vmac3 operation :

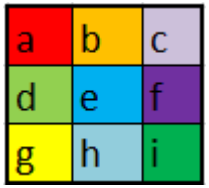
```
vacc =  (ushort32)v00 * 2 + (ushort32)v01 * 4;
vout = (uchar32)((vacc + (ushort32)v22 * 2 + (ushort32)v22 * 0) >> 5);
```

  - Store operation :

```
*(uchar32*)p_out_u8 = vout;
```

```
[slice  = 32 Bytes vector]
For (.. ) Horizontal loop {
        calculate input pointer of source slice
        calculate output pointer of destination slice
         For (.. ) Vertical loop        {
                read 9 input vectors
                calculate the filter (including scaling)
                store results
        }
}
```

Pixels Data



Vectors

Gaussian3X3 Vector Operators Code

Code Flow

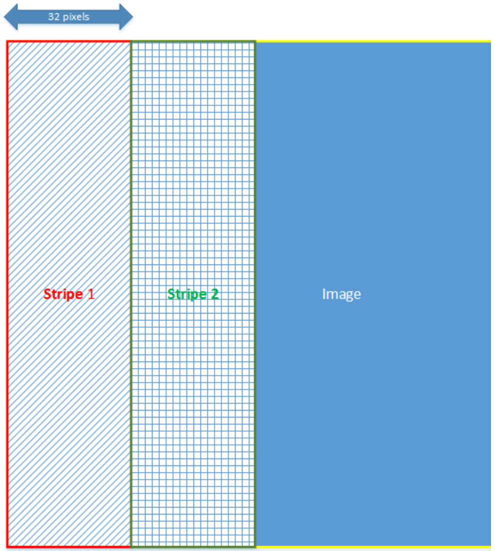
- The Kernel will work on the source buffer (size [M rows] x [N cols])
  ➔ In this example size = 128X64
- Scalar code process 9 pixels ➔ Vectorized code will process 9 vectors of 32 Bytes each.
- Outer loop will scan the buffer on the horizontal way (On cols). Inner loop on the Vertical (On rows)

Image Data

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119

Vectors Data

V00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
V01	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
V02	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
V10	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
V11	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73
V12	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74
V20	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92
V21	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93
V22	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94



Gaussian3X3 vector operators code – Cont.

- Loading the data
  - Each load instruction loads 32 bytes into one vector - **uchar32**
  - Each calculation produces 32 output points
  - The code flow performs 9 vector loads
    - Vector load example (3 loads out of 9)

```
uchar32 v00, v01, v02;
v00 = *(uchar32*)p_in_u8;
p_in_u8++;
v01 = *(uchar32*)p_in_u8;
p_in_u8++;
v02 = *(uchar32*)p_in_u8;
p_in_u8 += s32SrcStrideM2;
```

    - Note the different pointer post modification:
      - Advance to the next pixel
      - **Advance to the next pixel line**

Gaussian3X3 vector operators code – Cont.

- Gaussian calculation –filter implementation
  - We will use vector operators to perform the filter calculation
  - Vout = (V00 * 2 + V01 * 4 + V02 * 2 + V10 * 4 + V11 * 8 + V12 * 4 + V20* 2 + V21 * 4 + V22 * 2 + V22 * 0) >> 5;

```
uchar32 v00, v01, v02;

uchar32 v10, v11, v12;

uchar32 v20, v21, v22;

Uchar32 vout;

ushort32 vacc;

vacc  =  V00 * 2 + V01 * 4;

vacc += V02 * 2 + V10 * 4;

vacc += V11 * 8 + V12 * 4;

vacc += V20* 2 + V21 * 4;

vout = ((vacc +V22 * 2 + V22 * 0) >> 5);
```

2	4	2
4	8	4
2	4	2

V00	V01	V02
V10	V11	V12
V20	V21	V22

Gaussian3X3 vector operators code – Cont.

- Gaussian Calculation
  - Each Gaussian calculation contains 9 multiplications
  - By utilizing VMAC3 instructions 32 results can be achieved in 5 cycles
  - Calculation using multiple accumulate implementation code

```
uchar32 V00, V01, V10, V11;
ushort32 vacc;
vacc = (ushort32)V00* 2 + (ushort32)V01 * 4; //Line1 ➔ translated to vmad
vacc += (ushort32)V10 * 2 + (ushort32)V11 * 4;//Line2 ➔ translated to vmac3
```

```
VPU1.vmad {init} v23.uc32, r6.uc, v9.uc32, r4.uc, #0x0, v32.us16, v33.us16
VPU1.vmac3 v21.uc32, r6.uc, v22.uc32, r4.uc, v32.s16, v33.s16
```

  - What is the difference between line1 and line2, which operation line2 substitute?
  - Why vacc variable is defined as ushort32? is it accurate enough?

Gaussian3X3 vector operators code – Cont.

- Scaling and truncating the results from ushort ➔ uchar

```
uchar32 V22, vout;
ushort32 vacc;
vout =  (uchar32)((vacc+ (ushort32)V22 * 2 + (ushort32)V22 * 0) >> 5);
```

```
VPU0.vmac v3.uc32, #0x2, v34.s16, v35.s16, r7.uc, v14.c32
```

Gaussian3X3 Vector Operators Code – Cont.

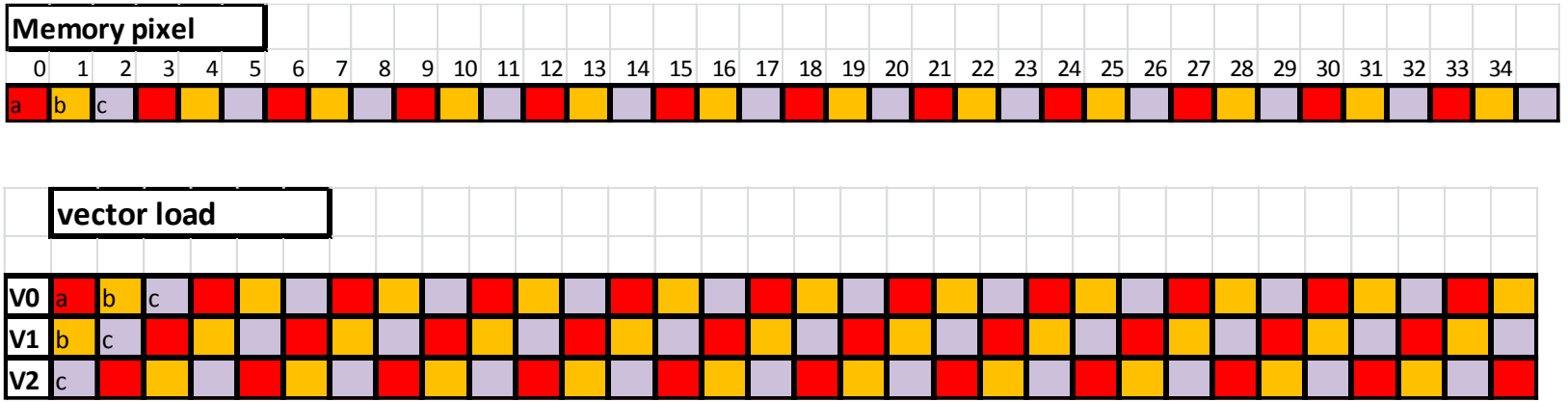
- Improvement suggestion
  - Each iteration includes:
    - 9 vectorized load instruction to load pixel data
    - 5 vmac3 instructions

- optimization suggestion
  - Reduce the number of load slots to get better balance with the vmac3 operations
  - Use of vldov vec-c intrinsic

Gaussian3X3 load Operation Optimizations

- Vldov instruction
  - Can load up to 4 vectors with a predefined offset of memory location using both LSU0 and LSU1
  - We will replace 3 vector loads with one vldov operation.
  - Vldov in this case will load 3 vectors with 3 horizontal positions with offset of one pixel

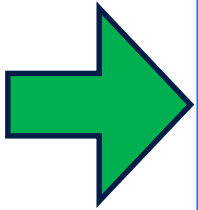
```
vldov((uchar32*)p_in_u8, v0, v1, v2, vdummy);
```



Gaussian3X3 load Operation Optimizations

- Substitution of regular loads by VLDOV
- Ensure that CEVA_VLDOV definition is defined in gaussian3x3_lib.h and other definitions are not defined.

```
v00 = *(uchar32*)p_in_u8;
p_in_u8++;
v01 = *(uchar32*)p_in_u8;
p_in_u8++;
v02 = *(uchar32*)p_in_u8;
p_in_u8 += s32SrcStrideM2;
v10 = *(uchar32*)p_in_u8;
p_in_u8++;
v11 = *(uchar32*)p_in_u8;
p_in_u8++;
v12 = *(uchar32*)p_in_u8;
p_in_u8 += s32SrcStrideM2;
v20 = *(uchar32*)p_in_u8;
p_in_u8++;
v21 = *(uchar32*)p_in_u8;
p_in_u8++;
v22 = *(uchar32*)p_in_u8;
p_in_u8 += s32SrcStrideP2;
```



```
vldov((uchar32*)p_in_u8, v00, v01, v02, vdummy);
p_in_u8 += s32SrcStep;
vldov((uchar32*)p_in_u8, v10, v11, v12, vdummy);
p_in_u8 += s32SrcStep;
vldov((uchar32*)p_in_u8, v20, v21, v22, vdummy);
p_in_u8 -= s32SrcStep;
```

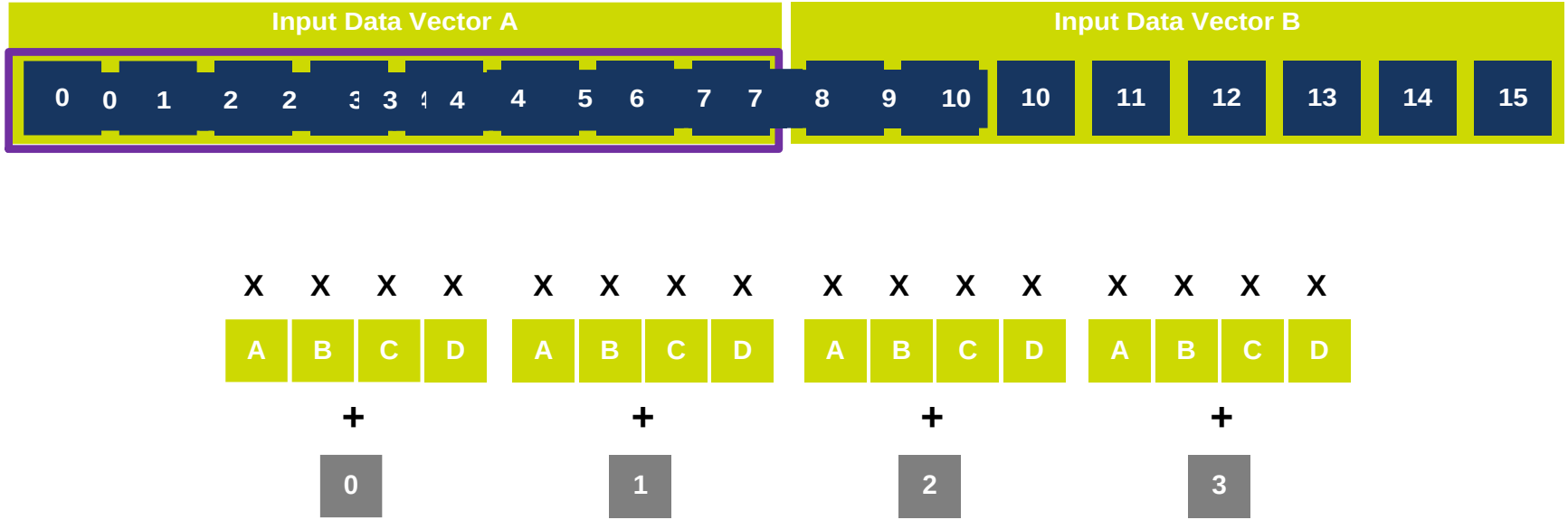
- Why do we define the last vector as dummy variable vector?

Agenda

- Glossary
- Median Filter 3X3
  - Algorithm Analysis
  - Efficient VEC-C Code
  - Performance
- **Gaussian3X3**
  - Gaussian3X3 Kernel Overview
  - Gaussian3X3 Vector Operators Code
  - Gaussian3X3 load Operation Optimizations
  - **Gaussian3X3 Sliding window code**

Gaussian3X3 Sliding Window Code

- The vector instruction performs the arithmetic operation multiple times in parallel (producing eight, 16 or 32 results). The next sliding-window operation is offset by a step from the previous operation.



Gaussian3X3 Sliding Window Code

- Loading vector coefficients

```
ushort coeff[16] = {2,4,2,0,4,8,4,0,2,4,2,0,0,0,0,0};
ushort16 v_coeff;
v_coeff = *(ushort16*)coeff;
```

- Loading pixel data

```
uchar32 v00,V10,V20;
uchar* p_in_u8  = (uchar *)&p_u8Src[i*16];
v00 = *(uchar32*)p_in_u8;
p_in_u8+=s32SrcStep;
v10 = *(uchar32*)p_in_u8;
p_in_u8+=s32SrcStep;
v20 = *(uchar32*)p_in_u8;
p_in_u8+=s32SrcStep;
```

Gaussian3X3 Sliding Window Code

- Sliding window intrinsic
- We will use three different type of sliding window operations
  - Sliding window multiply
  - Sliding window multiple accumulate
  - Sliding window multiple accumulating with scaling

Gaussian3X3 Sliding Window Code

- Sliding window intrinsic
- Vswmpy5 – serves also as initialization step of the accumulators
- **uint16 vswmpy5(uchar32 inA, uchar32 inB, ushort16 inC, unsigned int inD);**
  - **InA – 32 byte vector of loaded pixel data**
  - **InB – 32 byte vector of loaded pixel data**
  - **InC – 16 coefficients ➔ in Gaussian3X3 only 9 will be non-zero.**
  - **inD – offset and shift val control word**
  - **{inA , inB} X inC**
- **Example**

```
uchar32 v00;
ushort coeff[16] = {2,4,2,0, 4,8,4,0, 2,4,2,0, 0,0,0,0};
uint config0 = SW_CONFIG(0, 0, 0, 0, 0, 0);
uint16 vacc0;
vacc0 = (uint16)vswmpy5(v00, v00, v_coeff, config0);
```

Gaussian3X3 Sliding Window Code

- Sliding window intrinsic
- Vswmac5 – multiple accumulating
- uint16 vswmac5(accumulate_sw, uchar32 inA, uchar32 inB, ushort16 inC, unsigned int inD, uint16 inoutW);
  - **inA, InB, inC – inA,InB pixel input vectors inC – coefficient vector**
  - **inD – input, coefficient offset and accumulation post shift**
  - **inoutW – vector accumulator**
  - **accumulate_sw – accumulation switch – 'accumulate' is a predefined switch.**
- **Example**

```
uchar32 v10;
ushort coeff[16] = {2,4,2,0, 4,8,4,0, 2,4,2,0, 0,0,0,0};
uint config1 =  SW_CONFIG(0, 0, 0, 4, 0, 0); // 4 for coeff offset
uint16 vacc0;
vacc0 = vswmac5(accumulate,v10, v10, v_coeff, config1, vacc0);
```

Gaussian3X3 Sliding Window Code

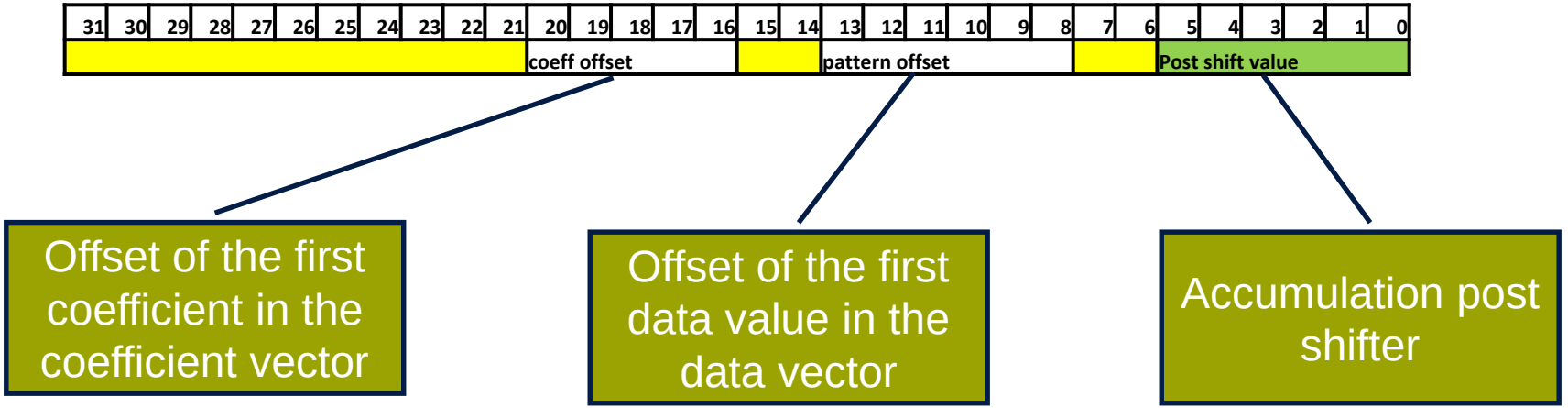
- Sliding window intrinsic:
- Vswmac5 accumulating and scaling
- short16 vswmac5(psl_sw, uchar32 inA, uchar32 inB, uchar32 inC, unsigned int inD, uint16 inE);
  - **inA, InB, inC – inA,InB pixel input vectors inC – coefficient vector**
  - **inD – input, coefficient offset and accumulation post shift**
  - **inE – vector accumulator**
  - **Psl_sw – post shifter switch, the value 'psl' is a predefined value.**
- Example

```
uchar32 v20;
Ushort16 vout;
ushort coeff[16] = {2,4,2,0, 4,8,4,0, 2,4,2,0, 0,0,0,0};
uint config2 = SW_CONFIG(5, 0, 0, 8, 0, 0); //5 for Post-shift (1/32)
vout= (ushort16)vswmac5(psl,v20, v20, v_coeff, config2, vacc0);
```

Gaussian3X3 Sliding Window Code

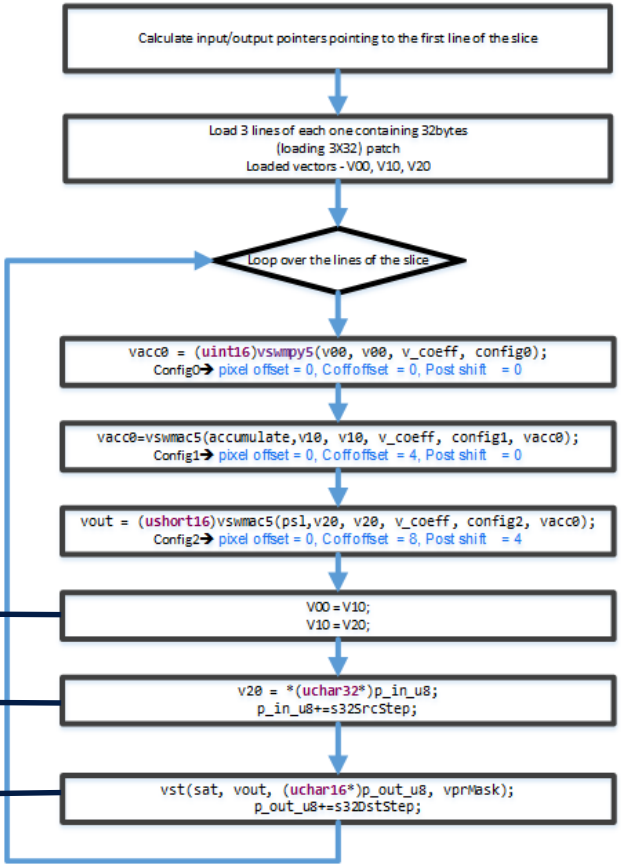
- Sliding window intrinsic:

- Control code field definition (configuration)
  - Control code is 32 bit width (assigned to one of the 32 scalar registers)



Gaussian3X3 Sliding Window Code

- Sliding window intrinsic:
- Iteration flow



Vector reuse between iterations

Read new pixel line

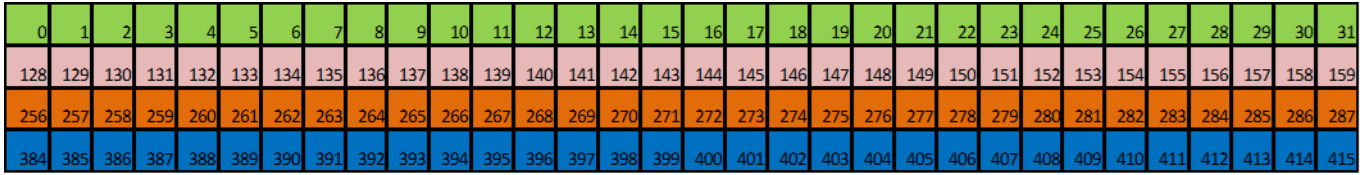
Store out results

inner loop

Outer loop

Gaussian3X3 Sliding Window Code

- Sliding window intrinsic:
- The algorithm will load 32 bytes on each iteration, but will store only16
- Preparation between inner/outer loops:
  - Update pointer of current input/output line to point to the begging of the slice
  - Read 32 pixels for consecutive lines into 3 vectors



- Use the different slide window instruction as specified in slides 45-48
- Reuse loaded vectors - each iteration in the inner loop should have only one load and one store.

Gaussian3X3 Sliding Window Code

- Run the application check bit-exact results and check the profiler report (ISS)

Profiler Statistics - Trace View

File Navigate Help

Function Viewer | Call Graph | Function Details | Branch View | Source View | Events

Name	Call count	Func size	Total (excl)	Avg (excl)	Total (incl)	Avg (incl)	Func level
No filters							Filtered
main	0	128	16	16	377250	377250	3
CCV_gaussian3x3_test(int)	1	756	128610	128610	276006	276006	3
CCV_gaussian3x3_ref(unsigned char*, unsigned char*, int, int, unsigned int, unsigned int)	1	416	107029	107029	107029	107029	3
fill_test_buffers_with_random_or_input_numbers()	1	682	100493	100493	100493	100493	3
CCV_gaussian3x3(unsigned char*, unsigned char*, int, int, unsigned int, unsigned int)	1	926	898	898	898	898	3
init_gaussian3x3_test(unsigned char*)	0	44	0	0	0	0	3
SimpleRNG_GetU32()	0	96	0	0	0	0	3

- Profile the same code using CAS Mode. What is the difference between ISS and CAS results?
- What is the theoretical cycles count? What is the compiler factor?

CEVA®
The DSP Powerhouse

THANK YOU

www.ceva-dsp.com