EVE Programming Environment

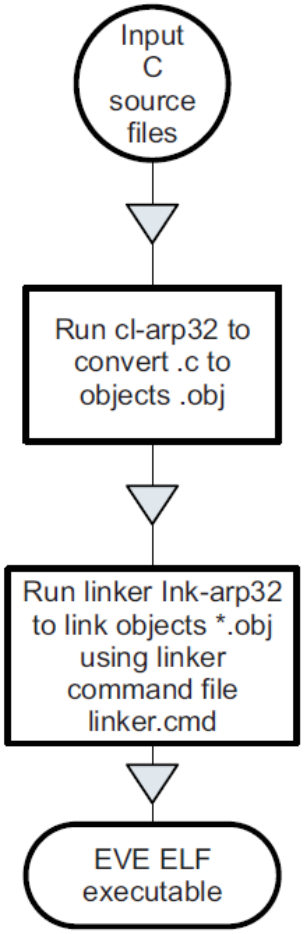
Contents

2D Block Addition for Unsigned Byte Arrays: Natural C Code

2D Block Addition for Unsigned Byte Arrays: Natural C Code

```
void vcop_vec_array_add_uns_char_cn
(
    unsigned char       a[],
    unsigned char       b[],
    unsigned char       c[],
    int                 blk_h,
    int                 blk_w
)
{
     int      i3;
     int      i4;

     for (i3 = 0; i3 < blk_h; i3++)
     {
         for (i4 = 0; i4 < blk_w; i4++)
         {
             c[blk_w * i3 + i4] =
             a[blk_w * i3 + i4] + b[blk_w * i3 + i4];
         }
     }
 }
```

Process to Build C files into Executable



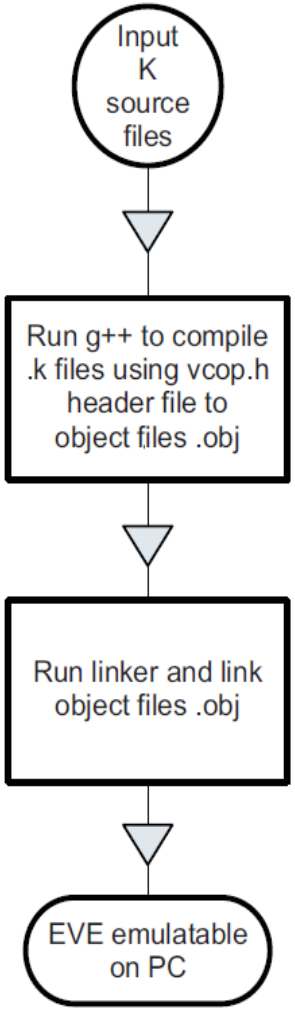
2D Block Addition for Unsigned Byte Arrays: VCOP Kernel-C Code

2D Block Addition for Unsigned Byte Arrays: VCOP Kernel-C Code

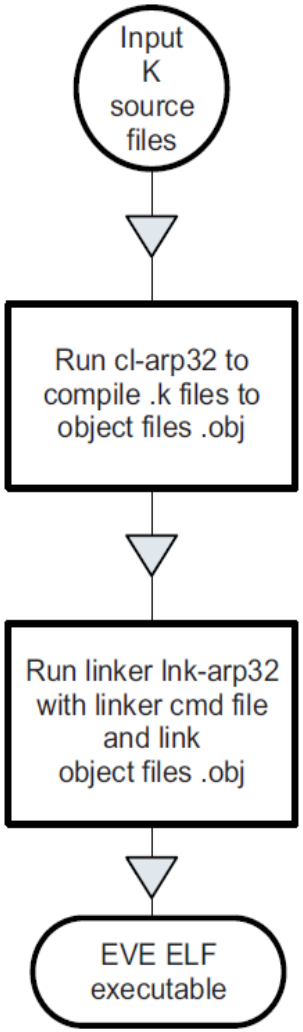
```
#define ELEMSZ          sizeof(*in1_ptr)
#define VECTORSZ        (VCOP_SIMD_WIDTH*ELEMSZ)


void eve_array_add_uns_char
(
   __vptr_uint8  in1_ptr,         // input 1 data pointer
   __vptr_uint8  in2_ptr,         // input 2 data pointer
   __vptr_uint8  optr,            // output data pointer
  unsigned short width,           // width of each line
  unsigned short height           // height of each line
)
{
   __vector Vin1;                 // input1
   __vector Vin2;                 // input2
   __vector Vout;                 // output


   for (int I1 = 0; I1 < height; I1++)
   {
       for (int I2 = 0; I2 < width/VCOP_SIMD_WIDTH; I2++)
       {
           __agen Addr;

           Addr = I1*width*ELEMSZ + I2*VECTORSZ;

           Vin1       = in1_ptr[Addr];
           Vin2       = in2_ptr[Addr];
           Vout       = Vin1 + Vin2;
           optr[Addr] = Vout;
       }
   }
}
```

Building the Vector Core Emulatable and Executable

PROCESS to BUILD EVE EMULATABLE



PROCESS to BUILD EVE EXECUTABLE



Various Components of Concurrent Scalar Vector Core Execution

Detailed View of Generated Interfaces for Kernel-C Functions

Interfaces Generated by Compiler for Every Kernel-C Function

```
/* Parameter Register Block                    */
extern unsigned short
__pblock_eve_array_add_uns_char[];

/* Basic Runner Function                       */
void eve_array_add_uns_char(
   __vptr_uint8 in1_ptr,
   __vptr_uint8 in2_ptr,
   __vptr_uint8 optr,
   unsigned short width,
   unsigned short height);

/* Custom Runner Function                      */
void eve_array_add_uns_char_custom(
   __vptr_uint8 in1_ptr,
   __vptr_uint8 in2_ptr,
   __vptr_uint8 optr,
   unsigned short width,
   unsigned short height,
   unsigned short* pblock);

/* Parameter Block Initialization Function     */
unsigned int eve_array_add_uns_char_init(
   __vptr_uint8 in1_ptr,
   __vptr_uint8 in2_ptr,
   __vptr_uint8 optr,
   unsigned short width,
   unsigned short height,
   unsigned short* pblock);

/* VCOP VLOOP Execution Function               */
void eve_array_add_uns_char_vloops(
   unsigned short* pblock);

/* Parameter Register Count Function           */
unsigned int
eve_array_add_uns_char_param_count();

/* Internal Value Count                        */
unsigned int
eve_array_add_uns_char_ctrl_count();
```

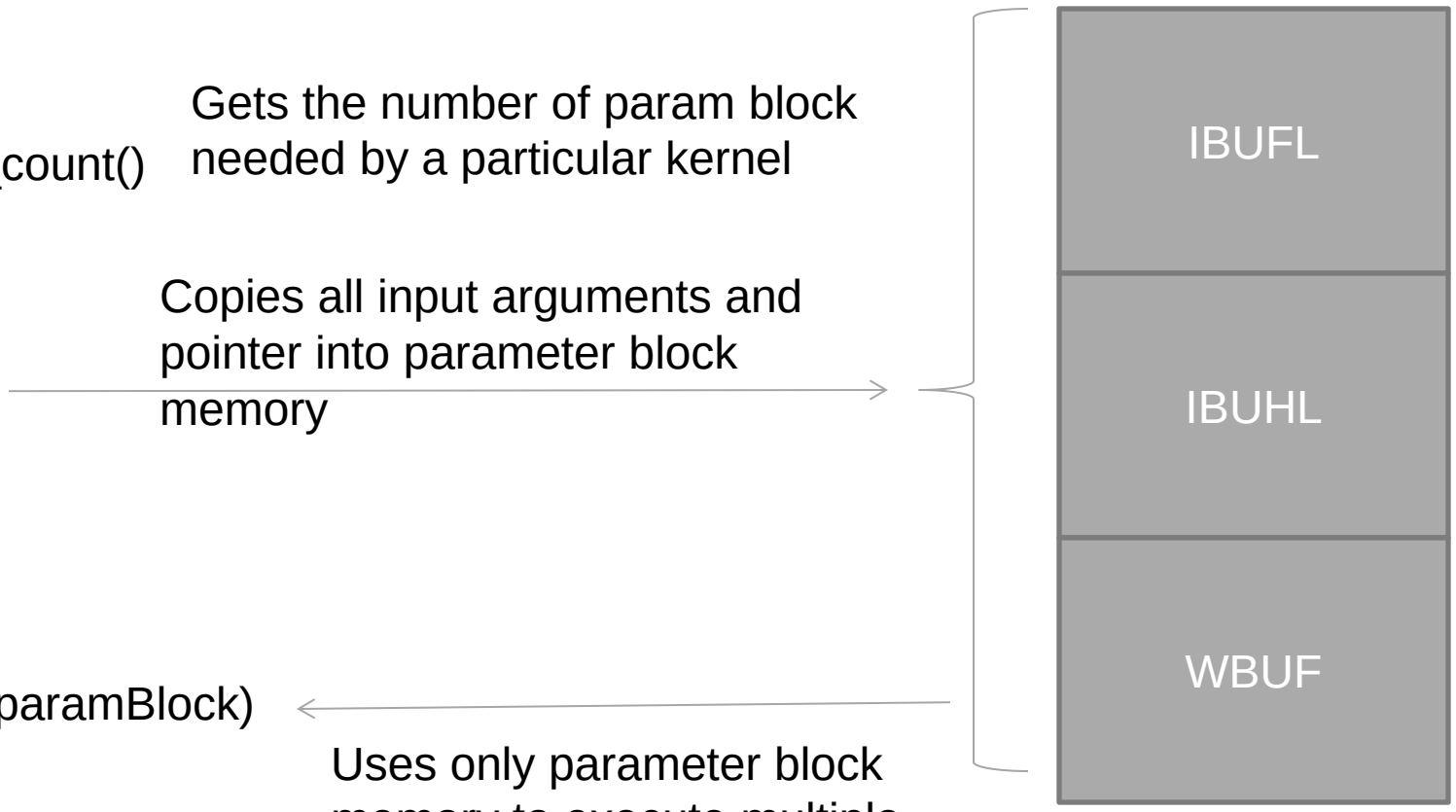
Parameter Block

_param_count()

Gets the number of param block needed by a particular kernel

_init()

Copies all input arguments and pointer into parameter block memory



_vloops(paramBlock)

Uses only parameter block memory to execute multiple number of times

Vector Core Loop Execution Time

- Command Decode Time: 2 x vector core command length + 4 = 2x6 + 4 = 16 cycles
- Parameter Fetch Time: 9 + ceiling((num-param_words/8)) = 9 + 1 = 10 cycles
- Loop execution Time: 16 cycles (pipeline ramp up/down) + (width x height)/8
- Total Time: (width x height)/8 + 41 (overhead) cycles

```
eve_array_add_uns_char_vloops:
80003c98            .text:eve_array_add_uns_char_vloops:
80003c98  00002a90    VCTRL R2, PARAM_PTR
80003c9c  00c60210    VLOOP COMP, CL#: 6, PL#: 6
80003ca0  00024814    VAGEN A0, [P8, P9, P0, P0]
80003ca4  000a4090    VLDBU_NPT P10[A0], V2
80003ca8  000c0090    VLDBU_NPT P12[A0], V0
80003cac  04080190    VADD V2, V0, V0
80003cb0  06060018    VSTBU_NPT_ALWS V0, P6[A0], RND_SAT: P0
80003cb4          01ff     RET
80003cb6          03ff    NOP
```

Vector Core Loop Execution Time

Table 2-4. Total Execution Time With Overheads With Exposed Command Decode

Memory	Width	Height	Execution time	Overheads	Total Time	Overhead%
64	8	8	8	41	49	83.67
1024	128	8	128	41	169	24.26
1024	32	32	128	41	169	24.26
2048	256	8	256	41	297	13.80
4096	512	8	512	41	553	7.41
4096	64	64	512	41	553	7.41
8192	1024	8	1024	41	1063	3.86
16384	128	128	2048	41	2089	1.96

Vector Core Loop Execution Time

Table 2-5. Total Execution Time With Overheads With Hidden Command Decode

Memory	Width	Height	Execution time	Overheads	Total Time	Overhead%
64	8	8	8	29	49	59.18
1024	128	8	128	29	169	17.16
1024	32	32	128	29	169	17.16
2048	256	8	256	29	297	9.76
4096	512	8	512	29	553	5.24
4096	64	64	512	29	553	5.24
8192	1024	8	1024	29	1063	2.73
16384	128	128	2048	29	2089	1.39

Vector Core Loop Execution Time from Simulator

- Iteration_Count: Product of four nested loop iteration counters (lpend1,.. lpend4)
- Operation_Count: Performance per iteration without memory stalls
- Load_store_cycle_count: Performance including impact of memory stalls.
- Cycle_count: Pipeline overhead of 16 cycles + Max (Load_store, Operation)
- Parameter_rd_overhead: Read time by vector core from memory for parameters
- Decode overhead: Loop decode overhead
- Total Cycle count = Sum of Decode overhead + Parameter_rd_overhead + Cycle_count
- Cumulative Cycle Count = Sum of Total cycle Count
- RPT_END : Maximum number of iterations for repeat loop

CONCLUSIONS

Conclusions

EVE Programming Environment

- 2D Block Addition for Unsigned Byte Arrays
  - Natural C Code
  - VCOP Kernel-C Code
- Building the Vector Core Emulatable and Executable
- Components of Concurrent Execution
- Generated Interfaces for Kernel-C Functions
- Vector Core Loop Performance
- Conclusions

Further Optimization

Contents

EVE Further Optimizations of Basic Examples

- 2D Block Addition for Unsigned Byte Arrays: More Efficient Code
- 2D Block Addition for Unsigned Short Arrays
- 2D Block Addition for Unsigned Short Arrays: More Efficient Code
- Restrictions: Alignment of Half-word data Loads
- Conclusions

2D Block Addition for Unsigned Bytes
(Code Shown in previous section)

```
#define ELEMSZ          sizeof(*in1_ptr)
#define VECTORSZ        (VCOP_SIMD_WIDTH*ELEMSZ)


void eve_array_add_uns_char
(
   __vptr_uint8  in1_ptr,         // input 1 data pointer
   __vptr_uint8  in2_ptr,         // input 2 data pointer
   __vptr_uint8  optr,            // output data pointer
  unsigned short width,           // width of each line
  unsigned short height           // height of each line
)
{
   __vector Vin1;                 // input1
   __vector Vin2;                 // input2
   __vector Vout;                 // output


   for (int I1 = 0; I1 < height; I1++)
   {
       for (int I2 = 0; I2 < width/VCOP_SIMD_WIDTH; I2++)
       {
           __agen Addr;

           Addr = I1*width*ELEMSZ + I2*VECTORSZ;

           Vin1       = in1_ptr[Addr];
           Vin2       = in2_ptr[Addr];
           Vout       = Vin1 + Vin2;
           optr[Addr] = Vout;
       }
   }
}
```


Performance: (width x height)/8 + 41 cycles

2D Block Addition for Unsigned Bytes (More Efficient Code)

```
#define ELEMSZ              sizeof(*in1_ptr)
#define VECTORSZ            (2 * VCOP_SIMD_WIDTH*ELEMSZ)
#define VCOP_2SIMD_WIDTH    (2 * VCOP_SIMD_WIDTH)

void eve_array_add_uns_char_intlv
(
   __vptr_uint8  in1_ptr,         // input 1 data pointer
   __vptr_uint8  in2_ptr,         // input 2 data pointer
   __vptr_uint8  optr,            // output data pointer
  unsigned short width,           // width of each line
  unsigned short height           // height of each line
)
{
   __vector Vin1,  Vin2;          // input from array input1
   __vector Vin3,  Vin4;          // input from array input2
   __vector Vout1, Vout2;         // outputs


   for (int I1 = 0; I1 < height; I1++)
   {
       for (int I2 = 0; I2 < width/VCOP_2SIMD_WIDTH; I2++)
       {
           __agen Addr;

          Addr = I1*width*ELEMSZ + I2*VECTORSZ;

          (Vin1,Vin2)      = in1_ptr[Addr].deinterleave();
          (Vin3,Vin4)      = in2_ptr[Addr].deinterleave();
          Vout1            = Vin1 + Vin3;
          Vout2            = Vin2 + Vin4;
          optr[Addr].interleave() = (Vout1, Vout2);
      }
   }
}
```


Performance: (width x height)/16 + 43 cycles

2D Block Addition for Unsigned Short Arrays

```
#define ELEMSZ           sizeof(*in1_ptr)
#define VECTORSZ        (VCOP_SIMD_WIDTH*ELEMSZ)

  void eve_array_add_uns_short
  (
     __vptr_uint16  in1_ptr,        // input 1 data pointer
     __vptr_uint16  in2_ptr,        // input 2 data pointer
     __vptr_uint16  optr,           // output data pointer
     unsigned short width,          // width of each line
     unsigned short height          // height of each line
  )
  {
     __vector Vin1;                 // input1
     __vector Vin2;                 // input2
     __vector Vout;                 // output

     for (int I1 = 0; I1 < height; I1++)
     {
         for (int I2 = 0; I2 < width/VCOP_SIMD_WIDTH; I2++)
       {
             __agen Addr;

             Addr       = I1*width*ELEMSZ + I2*VECTORSZ;

             Vin1       = in1_ptr[Addr];
             Vin2       = in2_ptr[Addr];
             Vout       = Vin1 + Vin2;
             optr[Addr] = Vout;
       }
     }
  }
```

Performance: (width x height)/8 + 41 cycles

2D Block Addition for Unsigned Short Arrays (More Optimized Code)

```
#define ELEMSZ              sizeof(*in1_ptr)
#define VECTORSZ            (2 * VCOP_SIMD_WIDTH*ELEMSZ)
#define VCOP_2SIMD_WIDTH    (2 * VCOP_SIMD_WIDTH)

void eve_array_add_uns_short_intlv
(
   __vptr_uint16  in1_ptr,         // input 1 data pointer
   __vptr_uint16  in2_ptr,         // input 2 data pointer
   __vptr_uint16  optr,            // output data pointer
   unsigned short width,           // width of each line
   unsigned short height           // height of each line
)
{
   __vector Vin1,  Vin2;          // input from array input1
   __vector Vin3,  Vin4;          // input from array input2
   __vector Vout1, Vout2;         // outputs


   for (int I1 = 0; I1 < height; I1++)
   {
       for (int I2 = 0; I2 < width/VCOP_2SIMD_WIDTH; I2++)
       {
           __agen Addr;

           Addr = I1*width*ELEMSZ + I2*VECTORSZ;

           (Vin1,Vin2)     = in1_ptr[Addr].deinterleave();
           (Vin3,Vin4)     = in2_ptr[Addr].deinterleave();
           Vout1           = Vin1 + Vin3;
           Vout2           = Vin2 + Vin4;
           optr[Addr].interleave() = (Vout1, Vout2);
       }
   }
}
```

Performance: (width x height)/16 + 43 cycles

EVE: Restrictions

Restriction 1—Vector Core Accessible Memories

The vector core cannot access ARP32 DMEM memory.

Restriction 2 — Half-Word INTLV/DINTLV Loads/Stores Must Be Word Aligned

2D Block Addition for Non-Aligned Unsigned Short Arrays

```
#define ELEMSZ              sizeof(*in1_ptr)
#define VECTORSZ            (2 * VCOP_SIMD_WIDTH*ELEMSZ)
#define VCOP_2SIMD_WIDTH    (2 * VCOP_SIMD_WIDTH)

void eve_array_add_uns_short_intlv
(
   __vptr_uint16 in1_ptr,         // input 1 data pointer
   __vptr_uint16 in2_ptr,         // input 2 data pointer
   __vptr_uint16 optr,            // output data pointer
  unsigned short width,           // width of each line
  unsigned short height           // height of each line
)
{
   __vector Vin1,  Vin2;          // input from array input1
   __vector Vin3,  Vin4;          // input from array input2
   __vector Vout1, Vout2;         // outputs

   for (int I1 = 0; I1 < height; I1++)
   {
       for (int I2 = 0; I2 < width/VCOP_2SIMD_WIDTH; I2++)
       {
           __agen Addr;

           Addr = I1*width*ELEMSZ + I2*VECTORSZ;

           Vin1 = in1_ptr[Addr];
           Vin2 = (in1_ptr + ELEMSZ*VCOP_SIMD_WIDTH)[Addr];

           Vin3 = in2_ptr[Addr];
           Vin4 = (in2_ptr + ELEM_SZ*VCOP_SIMD_WIDTH)[Addr];

           Vout1           = Vin1 + Vin3;
           Vout2           = Vin2 + Vin4;

           optr[Addr]                              = Vout1;
           (optr + ELEM_SZ*VCOP_SIMD_WIDTH)[Addr] = Vout2;
       }
   }
}
```


Performance: 2 x (width x height)/ 16 + 41 cycles

Conclusions

EVE Further Optimizations of Basic Examples

- 2D Block Addition for Unsigned Byte Arrays: More Efficient Code
- 2D Block Addition for Unsigned Short Arrays
- 2D Block Addition for Unsigned Short Arrays: More Efficient Code
- Restrictions: Alignment of Half-word data Loads
- Conclusions

EVE Examples of Simple Vector Core Features

Contents

EVE Further Optimizations of Basic Examples

- Example 1: 2D FIR Filter for 8-Bit Data
- Example 2: 2D FIR Filter for 16-bit Data
- Intelligent Read/Write Buffer
- Example 3: Complex Multiply
- Example 4: Searching for Array Maximum/Index
- List of VCOP Load and Store, Operation Modes
- Conclusions

Example 1: 2D FIR Filter for 8-Bit Data

2D FIR Filter for 8-bit data : Natural C Code

```
void vcop_fir_2D_uns_char_c
 (
    unsigned short          blk_w,
    unsigned short          line_ofst,
    unsigned short          blk_h,
    unsigned char           data_ptr[],
    unsigned short          coef_w,
    unsigned short          coef_h,
    unsigned char           coef_ptr[],
    unsigned char           output_ptr[],
    unsigned short          rnd_bits
 )
 {
    int  i1, i2, i3, i4;
    unsigned int  acc;
    unsigned char in_data, coef_data;

    for (i1 = 0; i1 < blk_h; i1++)
    {
        for (i2 = 0; i2 < blk_w; i2++)
        {
            acc = 0;

            for (i3 = 0; i3 < coef_h; i3++)
            {
                for (i4 = 0; i4 < coef_w; i4++)
                {
                    in_data     = data_ptr
                    [(i1 * line_ofst) + i2 + (i3 * line_ofst) + i4];

                    coef_data   = coef_ptr[(i3 * coef_w) + i4];

                    acc        += (in_data * coef_data);
                }
            }

            acc += (1 << (rnd_bits - 1));
            acc >>= rnd_bits;
            output_ptr[(i1 * line_ofst) + i2] = acc;
        }
    }
 }
```

2D FIR Filter for 8-bit data : VCOP Kernel-C Code

```
#define OUTPUT_VECTORSZ (VCOP_SIMD_WIDTH * OUTPUT_ELEMSZ)

void vcop_fir_2D_uns_char
(
    unsigned short  blkw,        // width of input block, in elements
    unsigned short  line_ofst,   // offset between input lines, in elems
    unsigned short  blkh,        // height of input block
    __vptr_uint8    data_ptr,    // input data pointer
    unsigned short  coefw,       // width of coef block, in elements
    unsigned short  coefh,       // height of coef block
    __vptr_uint8    coef_ptr,    // coef data pointer
    __vptr_uint8    output_ptr, // output data pointer
    unsigned short  rnd_bits     // bit position for rounding
)
{
   __vector Vin, Vcoef, Vout;

   for (int I1 = 0; I1 < blkh; I1++)
   {
      for (int I2 = 0; I2 < (blkw/VCOP_SIMD_WIDTH); I2++)
      {
         __agen A2;
         Vout = 0;

         for (int I3 = 0; I3 < coefh; I3++)
         {
            for (int I4 = 0; I4 < coefw; I4++)
            {
               __agen A0, A1;
               A0  = I1*LINESZ    + I2*VECTORSZ +
                     I3*LINESZ    + I4*ELEMSZ;

               Vin = data_ptr[A0];
               A1    = I3*COEF_LINESZ + I4*COEF_ELEMSZ;
               Vcoef = coef_ptr[A1].onept();
               Vout += Vin * Vcoef;
            }
         }
         A2 = I1*(OUTPUT_LINESZ) + I2*(OUTPUT_VECTORSZ);
         output_ptr[A2] = Vout.round(rnd_bits);
      }
   }
```

Performance: (coefw x coefh x blkw x blkh)/8 + 48 cycles

2D FIR Filter for 8-bit: Further Optimizations

```
#define ELEMSZ              sizeof(*data_ptr)
#define VCOP_2SIMD_WIDTH    (2 * VCOP_SIMD_WIDTH)
#define LINESZ              (lofst*ELEMSZ)
#define VECTORSZ            (VCOP_2SIMD_WIDTH * ELEMSZ)
#define COEF_ELEMSZ         sizeof(*coef_ptr)
#define COEF_LINESZ         (coefw*COEF_ELEMSZ)
#define OUTPUT_ELEMSZ       sizeof(*output_ptr)
#define OUTPUT_LINESZ       (lofst*OUTPUT_ELEMSZ)
#define OUTPUT_VECTORSZ     (VCOP_2SIMD_WIDTH * OUTPUT_ELEMSZ)

void eve_fir2d
(
    unsigned short blkw,          // width of input block, in elements
    unsigned short lofst,         // offset between input lines, elems
    unsigned short blkh,          // height of input block
    unsigned short vds,           // vertical downsample factor
    __vptr_uint8   data_ptr,      // input data pointer
    Uint16         coefw,         // width of coef block, in elements
    Uint16         coefh,         // height of coef block
    __vptr_uint8   coef_ptr,      // coef data pointer
    __vptr_uint8   output_ptr,    // output data pointer
   Uint16          rnd_bits       // bit position for rounding
)
{
   __vector   Vin1,  Vin2, Vcoef, Vout1, Vout2;

   for (int I1 = 0; I1 < blkh/vds; I1++)
   {
      for (int I2 = 0; I2 < (blkw/VCOP_2SIMD_WIDTH); I2++)
      {
         __agen A2;
         Vout1 =  0;  Vout2 = 0;

         for (int I3 = 0; I3 < coefh; I3++)
         {
            for (int I4 = 0; I4 < coefw; I4++)
            {
               __agen A0, A1;
               A0 = I1*LINESZ*vds +I2*VECTORSZ +  I3*LINESZ +
                    I4*ELEMSZ;
               A1 = I3*COEF_LINESZ + I4*COEF_ELEMSZ;
              (Vin1, Vin2) = data_ptr[A0].deinterleave();
               Vcoef = coef_ptr[A1].onept();
               Vout1 += Vin1 * Vcoef;
               Vout2 += Vin2 * Vcoef;
            }
         }
         A2 = I1*(OUTPUT_LINESZ) + I2*(OUTPUT_VECTORSZ);
         output_ptr[A2].interleave() = (Vout1, Vout2).round(rnd_bits);
      }
   }
}
```

Performance:
(coefw x coefh x blkw x blkh)/16
+ 53 cycles

Example 2: 2D FIR Filter for 16-bit Data

2D FIR Filter for 16-bit

```
#define ELEMSZ              sizeof(*data_ptr)
#define LINESZ              (lofst*ELEMSZ)
#define LINE2SZ             (2 * lofst*ELEMSZ)
#define VECTORSZ            (VCOP_SIMD_WIDTH * ELEMSZ)
#define COEF_ELEMSZ         sizeof(*coef_ptr)
#define COEF_LINESZ         (coefw*COEF_ELEMSZ)
#define OUTPUT_ELEMSZ       sizeof(*output_ptr)
#define OUTPUT_LINE2SZ      (2 * lofst * OUTPUT_ELEMSZ)
#define OUTPUT_VECTORSZ     (VCOP_SIMD_WIDTH * OUTPUT_ELEMSZ)
void eve_fir2d
(
   unsigned short blkw,         // width of input block, in elements
   unsigned short lofst,        // offset between input lines, in elems
   unsigned short blkh,         // height of input block
   unsigned short vds,          // vertical downsample factor
   __vptr_int16   data_ptr,     // input data pointer
   unsigned short coefw,        // width of coef block, in elements
   unsigned short coefh,        // height of coef block
   __vptr_int16   coef_ptr,     // coef data pointer
   __vptr_int16   output_ptr,   // output data pointer
   unsigned short rnd_bits      // bit position for rounding
)
{
  __vector Vin1,  Vin2, Vcoef, Vout1, Vout2;

  for (int I1 = 0; I1 < blkh/2; I1++)
  {
     for (int I2 = 0; I2 < (blkw/VCOP_SIMD_WIDTH); I2++)
     {
        __agen A2;
        Vout1  =   0;
        Vout2  =   0;

        for (int I3 = 0; I3 < coefh; I3++)
        {
           for (int I4 = 0; I4 < coefw; I4++)
           {
              __agen A0, A1;
              A0  = I1*LINE2SZ*vds + I2*VECTORSZ +
                    I3*LINESZ      + I4*ELEMSZ;
              A1  = I3*COEF_LINESZ + I4*COEF_ELEMSZ;

              Vin1 = data_ptr[A0].npt();
              Vin2 = (data_ptr + LINESZ)[A0].npt();
              Vcoef = coef_ptr[A1].onept();
              Vout1 += Vin1 * Vcoef;
              Vout2 += Vin2 * Vcoef;
           }
        }

        A2 = I1*(OUTPUT_LINE2SZ) + I2*(OUTPUT_VECTORSZ);
        output_ptr[A2] = Vout1.round(rnd_bits);
        (output_ptr + LINESZ)[A2] = Vout2.round(rnd_bits);
     }
  }
}
```

Performance:

1.125 x (coefw x coefh x blkw x blkh)/16
+ 53 cycles

Intelligent Read/Write Buffer

Intelligent Read Buffer

- Load Unit Buffer

➢ Each memory access always loads 256-bits into the load buffer.

➢ Tries to re-use load buffer data for a subsequent load request.

➢ Can return either 64/128 bits if it falls completely within buffer.

➢ If all 8 load buffers hit, instantaneous bandwidth = 1024 bits

➢ Sustained load bandwidth = 768 bits

Iteration	Data Needed							
1	X00	X01	X02	X03	X04	X05	X06	X07
2	X01	X02	X03	X04	X05	X06	X07	X08
3	X02	X03	X04	X05	X06	X07	X08	X09
4	X10	X11	X12	X13	X14	X15	X16	X17
5	X11	X12	X13	X14	X15	X16	X17	X18
6	X12	X13	X14	X15	X16	X17	X18	X19
7	X08	X09	X0A	X0B	X0C	X0D	X0E	X0F

➢ Iteration 1 = {X00,…….X07} and {X10,……X17}
➢ Iteration 2: Re-use
➢ Iteration 3: Re-use

➢Performance = {2, 1, 1, 2, 1, 1} = 4/3 = 1.33 cycles

Intelligent Write Buffer

- Delays writes giving priority to reads by using write buffer.
- Allows writes to drain out by giving higher priority to load.
- Previous example has 2 stores, writes get queued.
- Writes get committed to memory in cycles where there are no loads.
- Hardware capability does not require programmer to intervene.

List of VCOP Load and Store, Operation Modes

List Of Operations

Operation	#in-out	#bits	Kernel-C Expression [1]
VABS	1-1	40	Vdst = abs(Vsrc);
VABSDIF	2-1	40	Vdst = abs(Vsrc1 – Vsrc2);
VADD	2-1	40	Vdst = Vsrc1 + Vsrc2;
VADD3	3-1	40	Vdst += Vsrc1 + Vsrc2;
VADDH	2-1	40	Vdst = Vsrc1 + hi(Vsrc2);
VADIF3	3-1	40	Vdst = Vsrc3 + (Vsrc1 – Vsrc2)
VAND	2-1	40	Vdst = Vsrc1 & Vsrc2;
VAND3	3-1	40	Vdst &= Vsrc1 & Vsrc2;
VANDN	2-1	40	Vdst &= ~Vsrc;
VADDSUB	2-2	40	(Vsrc1, Vsrc2).addsub();
VBINLOG	1-1	32	Vdst = binlog(Vsrc);
VBITC	1-1	32	Vdst = count_bits(Vsrc);
VBITDI	1-2	32	(Vdst1, Vdst2) = deinterleave_bits(Vsrc);
VBITI	2-1	32	Vdst = interleave_bits(Vsrc1, Vsrc2);
VBITPK	2-1	33	Vdst = pack(Vsrc1 >= Vsrc2);
VBITR	1-1	32	Vdst = reverse_bits(Vsrc);
VBITTR	1-1	NSIMD	Vdst = transpose_bits(Vsrc);
VBITUNPK	2-1	33	Vdst = unpack(Vsrc1, Vsrc2);
VCMOV		40	Vdst = Vsrc;
VCMPEQ	2-1	40	Vdst = Vsrc1 == Vsrc2;
VCMPGE	2-1	40	Vdst = Vsrc1 >= Vsrc2;
VCMPGT	2-1	40	Vdst = Vsrc1 > Vsrc2;
VDINTRLV	2-2	40	(Vsrc1, Vsrc2).deinterleave();
VDINTRLV2	2-2	40	(Vsrc1, Vsrc2).deinterleave2();
VEXITNZ		40	--
VINTRLV	2-2	40	(Vsrc1, Vsrc2).interleave();
VINTRLV2	2-2	40	(Vsrc1, Vsrc2).interleave2();
VINTRLV4	2-2	40	(Vsrc1, Vsrc2).interleave4();
VLMBD	3-1	40	Vdst = leading_bit(Vsrc1, Vsrc2);

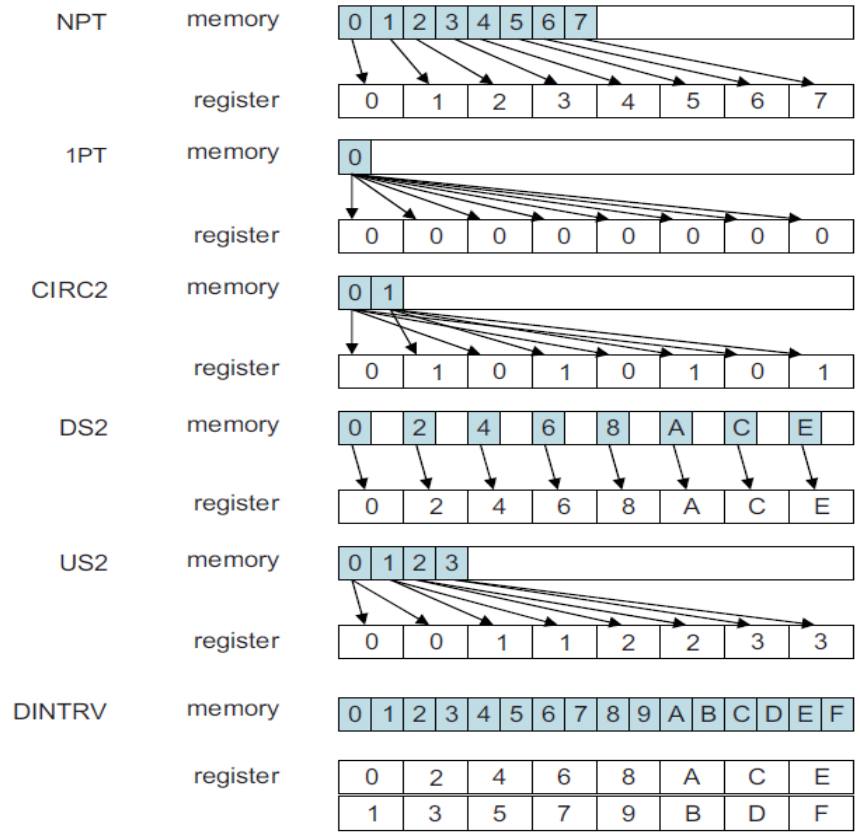
[1] K = 8, 15, 16 are shift amounts supported by multiplier

List Of Operations

Operation	#in-out	#bits	Kernel-C Expression [1]
VMADD	3-1	17/40	Vdst += Vsrc1 × Vsrc2;
VMADD	3-1	17/40	Vdst += (Vsrc1 × Vsrc2).round(K);
VMADD	3-1	17/40	Vdst += (Vsrc1 × Vsrc2).truncate(K);
VMADD	3-1	17/40	Vdst += Vsrc1 × Vsrc2 << 1;
VMAX	2-1	33	Vdst = max(Vsrc1, Vsrc2);
VMAXSETF	2-2	33	(Vsrc2, Vdst2) = maxf(Vsrc1,Vsrc2)
VMIN	2-1	33	Vdst = min(Vsrc1, Vsrc2);
VMINSETF	2-2	33	(Vsrc2, Vdst2) = minf(Vsrc1,Vsrc2)
VMPY	2-1	17/33	Vdst = Vsrc1 × Vsrc2;
VMPY	2-1	17/33	Vdst = (Vsrc1 × Vsrc2).round(K);
VMPY	2-1	17/33	Vdst = (Vsrc1 × Vsrc2).truncate(K);
VMPY	2-1	17/33	Vdst = (Vsrc1 × Vsrc2) << 1;
VMSUB	3-1	17/40	Vdst -= Vsrc1 × Vsrc2;
VMSUB	3-1	17/40	Vdst -= (Vsrc1 × Vsrc2).round(K);
VMSUB	3-1	17/40	Vdst -= (Vsrc1 × Vsrc2).truncate(K);
VMSUB	3-1	17/40	Vdst -= Vsrc1 × Vsrc2 << 1;
VNOP		40	-
VNOT	1-1	40	Vdst = ~Vsrc;
VOR	2-1	40	Vdst = Vsrc1 | Vsrc2;
VOR3	3-1	40	Vdst |= Vsrc1 | Vsrc2;
VRND	2-1	40/5	Vdst = round(Vsrc1, Vsrc2);
VSAD	3-1	40	Vdst += abs(Vsrc1 – Vsrc2);
VSEL	3-1	40	Vdst = select(Vsrc1, Vsrc2, Vdst);
VSHF	2-1	40/6	Vdst = Vsrc1 << Vsrc2;
VSHF16	1-2	40	(Vdst1, Vdst2) = jus16(Vsrc1)
VSHFOR	3-1	40/6	Vdst |= Vsrc1 << Vsrc2;
VSIGN	2-1	40	Vdst = apply_sign(Vsrc1, Vsrc2);
VSORT2	2-2	33	(Vsrc1, Vsrc2).minmax();
VSUB	2-1	40	Vdst = Vsrc1 - Vsrc2;
VSWAP	3-2	40	(Vsrc1, Vsrc2).swap(Vcond)
VXOR	2-1	40	Vdst = Vsrc1 ^ Vsrc2;

List of Load Modes

Load Mode	Kernel-C Expression
Vector Load	Vreg = <pexpr>[Agen].npt();
Scalar Load	Vreg = <pexpr>[Agen].onept();
Cicrcular-2	Vreg = <pexpr>[Agen].circ2();
Down Sample by 2	Vreg = <pexpr>[Agen].ds2();
Up Sample by 2	Vreg = <pexpr>[Agen].us2();
Custom Distribution (compile time)	<pexpr>[Agen].dist(K1,K2,K3,K4,K5,K6,K7,K8);
Custom Distribution (run time)	Vreg = <pexpr>[Agen].dist(<pexpr>);
De-Interleave	(Vreg1, Vreg2) = <pexr>[Agen].deinterleave()
N-bit load to N-way	Vreg = <pexpr>[Agen].Nbits();



List of Store Modes

Store Mode	Kernel-C Expression
Vector Store	<pexpr>[Agen].npt() = Vreg;
Scalar Store	<pexpr>[Agen].onept() = Vreg;
Down-Sample by 2.	<pexpr>[Agen].ds2() = Vreg;
Tranpsose (Offset_NP1)	<pexpr>[Agen].offset_np1 = Vreg;
Scatter	<pexpr>[Agen].scatter(Vindex) = Vreg;
Interleaved	<pexpr> [Agen] = (Vreg1,Vreg2).interleave() ;
Collated	<pexpr> [collate(Vpred)] = Vsrc;
Skip every other element (B/H only)	<pexpr>[Agen].skip = Vreg

Distribution	vreg[r][0] goes to	vreg[r][1] goes to	....	vreg[r][5] goes to	vreg[r][6] goes to	vreg[r][7] goes to
NPT	dptr[0]	dptr[1]	...	dptr[5]	dptr[6]	dptr[7]
1PT	dptr[0]	n/a		n/a	n/a	n/a
DS2	dptr[0]	n/a	...	n/a	dptr[3]	n/a
INTRLV	dptr[0] = vreg[r][0], dptr[1] = vreg[r+1][0]	dptr[2] = vreg[r][1], dptr[3] = vreg[r+1][1]	...	dptr[10] = vreg[r][5], dptr[11] = vreg[r+1][5]	dptr[12] = vreg[r][6], dptr[13] = vreg[r+1][6]	dptr[14] = vreg[r][7], dptr[15] = vreg[r+1][7]
OFFST_NP1	dptr[0]	dptr[9]	...	dptr[45]	dptr[54]	dptr[63]
COLLAT	none or *dptr++	none or *dptr++	...	none or *dptr++	none or *dptr++	none or *dptr++
DDA	dptr[V0[0]]	dptr[V0[1]]	...	dptr[V0[5]]	dptr[V0[6]]	dptr[V0[7]]
SKIP	dptr[0]	dptr[2]	...	dptr[10]	dptr[12]	dptr[14]

CONCLUSIONS

Conclusions

- Example 1: 2D FIR Filter for 8-Bit Data
- Example 2: 2D FIR Filter for 16-bit Data
- Intelligent Read/Write Buffer
- Example 3: Complex Multiply
- Example 4: Searching for Array Maximum/Index
- List of VCOP Load and Store, Operation Modes
- Practice by writing a lot of basic examples
- Start off by re-building existing examples