MSOC

Nov. 28, 2020

Synthesizable H.264/AVC Decoder

Presenter: Ting-Yung Chen, Yu-Cheng Lin, I-Hsuan Liu

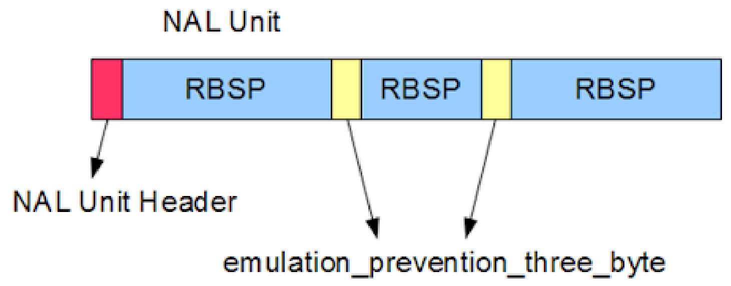
Team#: 1

Outline

- Introduction to H.264/AVC
- About this project
- Challenges and Solutions
- Insights
- Future work

About H.264/AVC

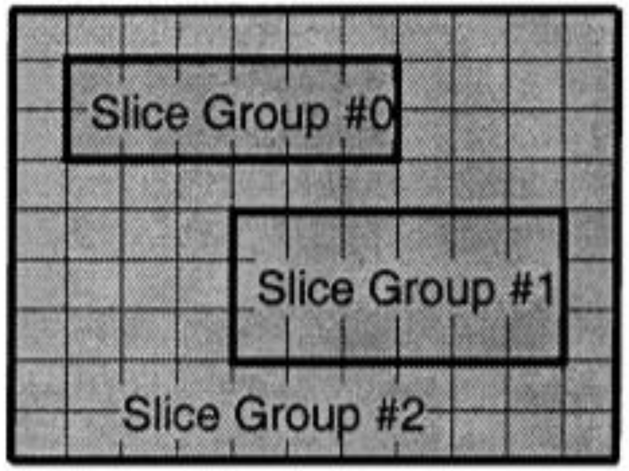
- Network Abstraction Layer (NAL) Unit
  - Effective packet contains integer number of bytes
- Data Storage
  - YCbCr 4:2:0
  - Y:Luma, CbCr: Chroma
- Macroblock (16x16 pixels)
- Subblock (4x4 pixels)
  - Some can be calculated by neighbor (temporal/spatial) blocks



NALU

Slice

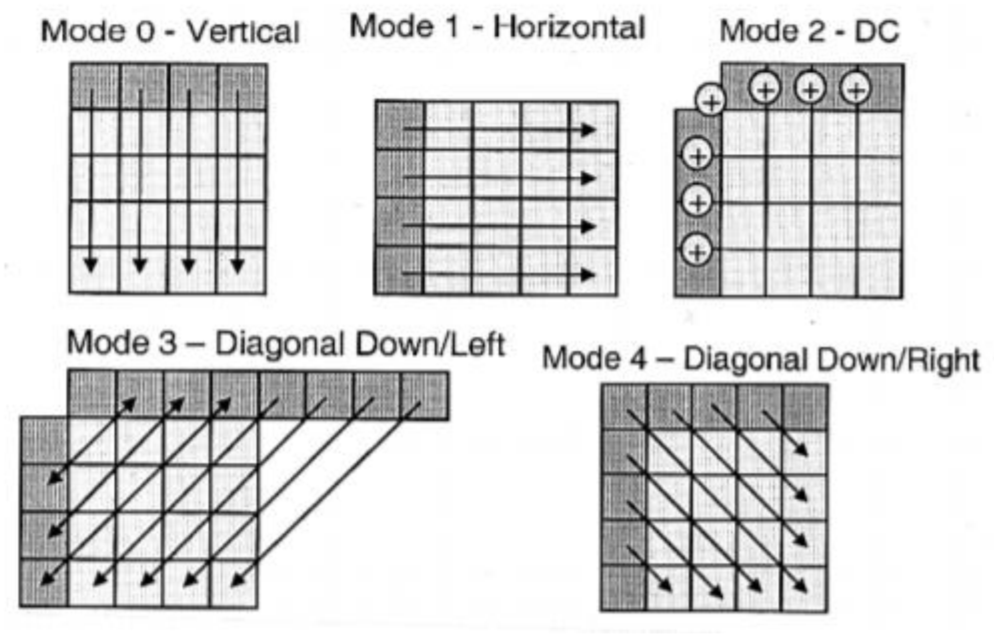
- Each frame can be divided into multiple slices, which contains several macroblocks
- Each slice is independent
- Macroblock prediction mode
  - Intra (spatial)
  - Inter (temporal)
  - I_PCM (skip)
- There are 3 type of slices
  - I slice: intra only
  - P slice: intra, 1-step-mvp inter
  - B slice: intra, 1/2-step-mvp inter



Slice Groups

Intra-Prediction

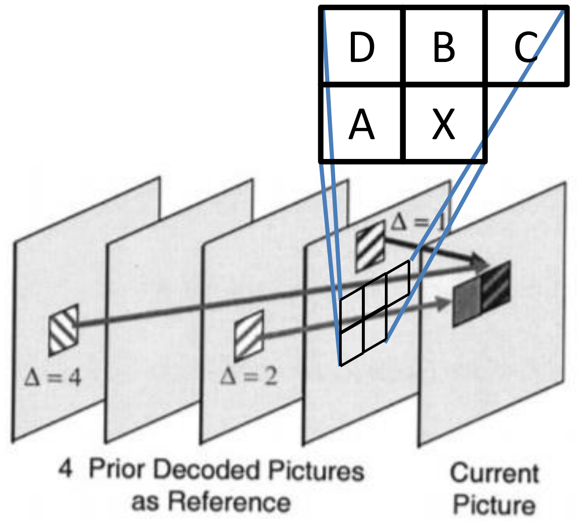
- 9 modes for Luma prediction (16x16 pixels)
- 4 modes for Chroma prediction (8x8 pixels)



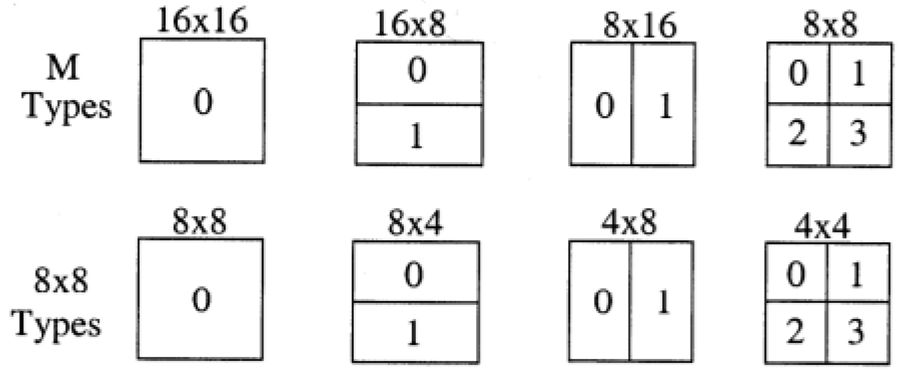
Intra-prediction modes

Inter-prediction

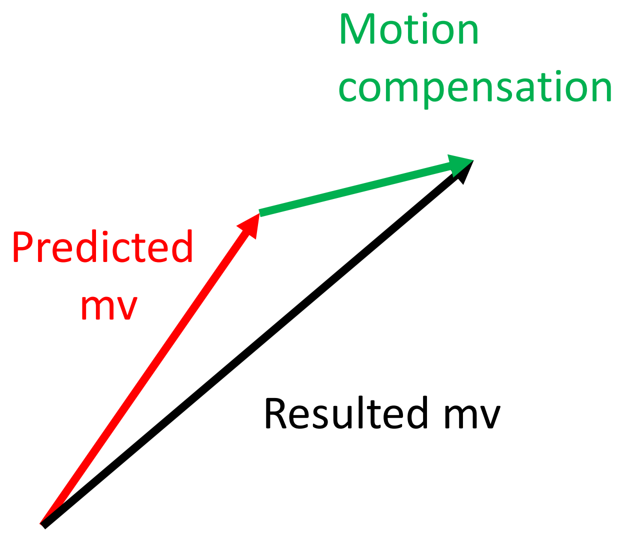
- Predict motion vector based on neighbor blocks of the reference frames
- Each Macro-block has 16 motion vectors
- Each 4x4 block have its own motion vector
- Motion compensation for each block segmentation



Motion vector predictor



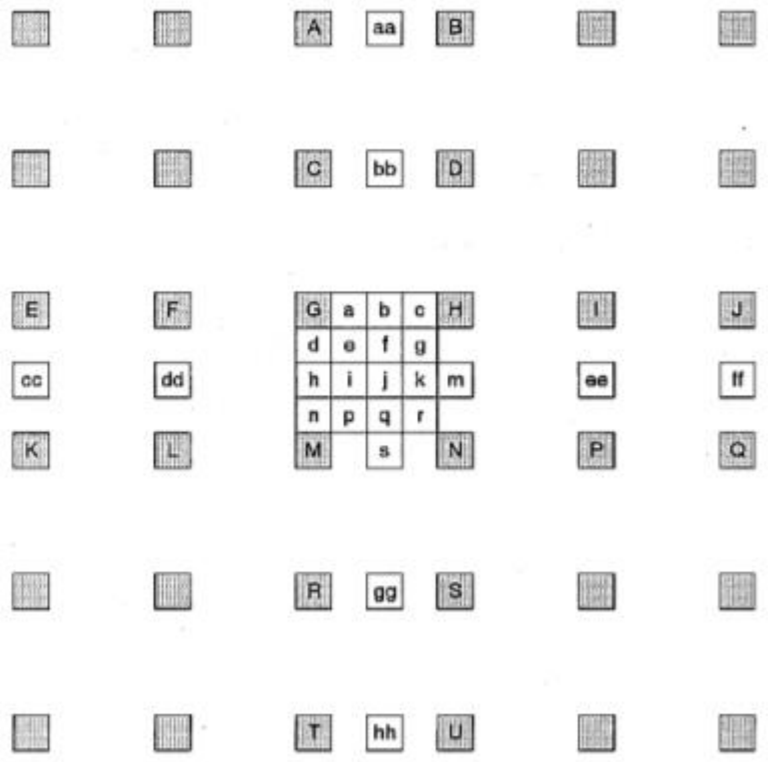
Block Segmentations



Resulted mv

Inter-prediction

- Interpolation for fraction motion vector



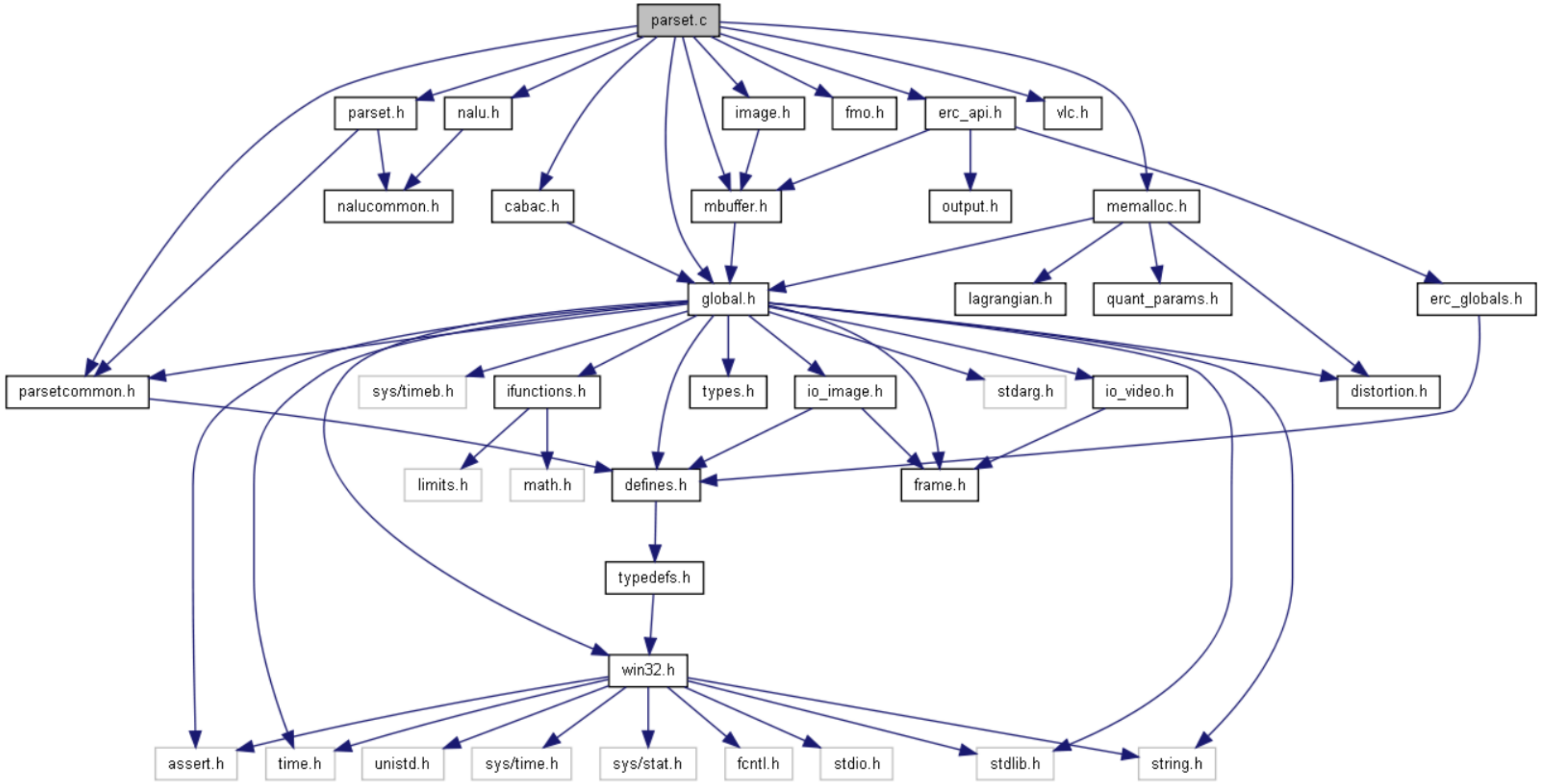
Integer sampling ref. frame

$$b_1 = (E - 5F + 20G + 20H - 5I + J)$$
$$h_1 = (A - 5C + 20G + 20M - 5R + T)$$
$$b = (b_1 + 16) \gg 5$$
$$h = (h_1 + 16) \gg 5.$$
$$j_1 = cc - 5dd + 20h_1 + 20m_1 - 5ee + ff$$
$$a = (G + b + 1) \gg 1.$$
$$e = (b + h + 1) \gg 1.$$

Interpolation formula

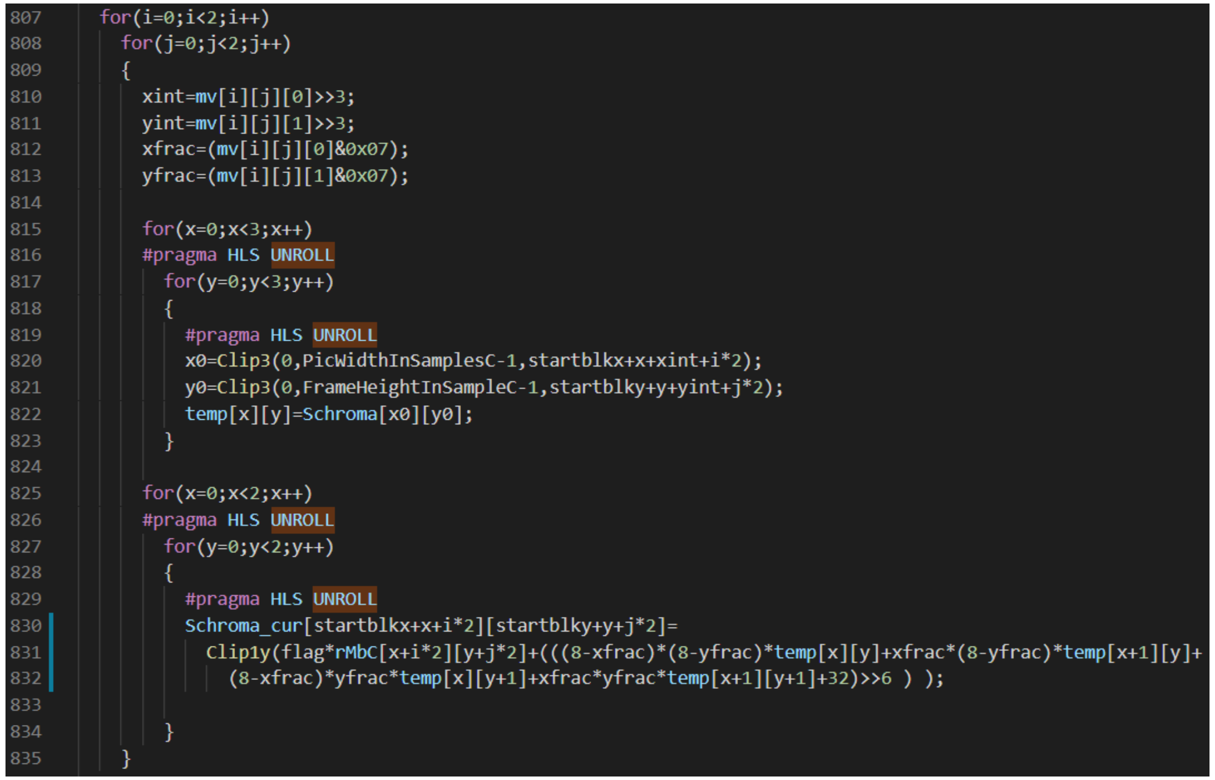
About this project

- H.264/AVC JM16.0 Reference Software Decoder is a open source code with detail documentation
- This project aims to make this software synthesizable

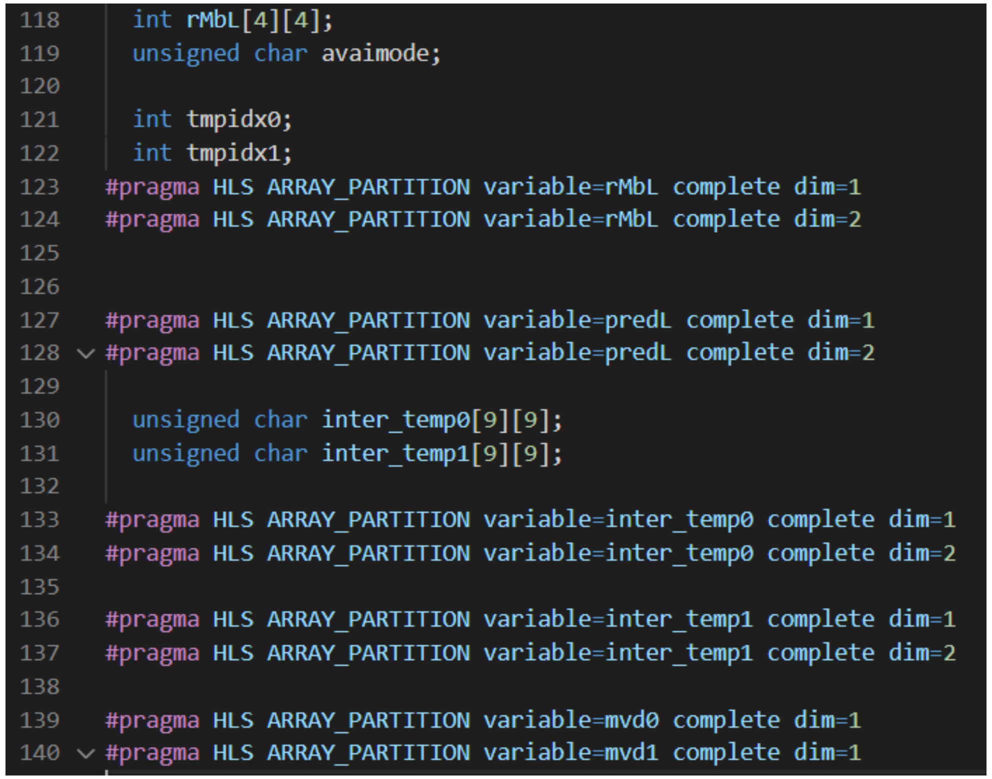


About this project

- What the author has done is simply
  - Add UNROLL pragma for every for loop that contains array data type
  - Add ARRAY_PARTITION for every internal memories
  - Add PIPELINE for some remaining for loop



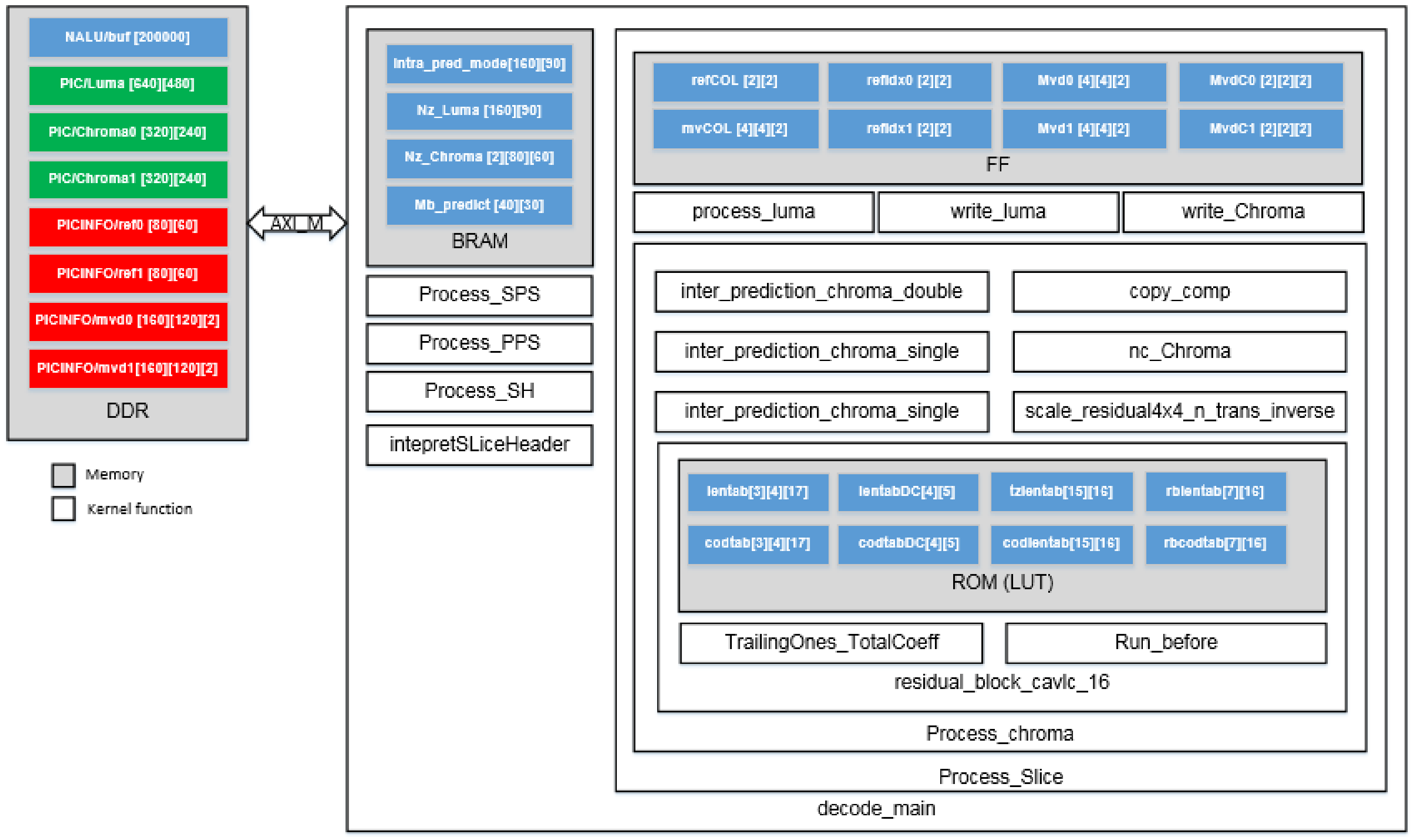
```
for(i=0;i<2;i++)
  for(j=0;j<2;j++)
  {
    xint=mv[i][j][0]>>3;
    yint=mv[i][j][1]>>3;
    xfrac=(mv[i][j][0]&0x07);
    yfrac=(mv[i][j][1]&0x07);

    for(x=0;x<3;x++)
    #pragma HLS UNROLL
      for(y=0;y<3;y++)
      {
        #pragma HLS UNROLL
        x0=Clip3(0,PicWidthInSamplesC-1,startblkx+x+xint+i*2);
        y0=Clip3(0,FrameHeightInSampleC-1,startblky+y+yint+j*2);
        temp[x][y]=Schroma[x0][y0];
      }

    for(x=0;x<2;x++)
    #pragma HLS UNROLL
      for(y=0;y<2;y++)
      {
        #pragma HLS UNROLL
        Schroma_cur[startblkx+x+i*2][startblky+y+j*2]=
          Clip1y(flag*rMbC[x+i*2][y+j*2]+(((8-xfrac)*(8-yfrac)*temp[x][y]+xfrac*(8-yfrac)*temp[x+1][y]+
            (8-xfrac)*yfrac*temp[x][y+1]+xfrac*yfrac*temp[x+1][y+1]+32)>>6 ) );

      }
  }
```



```
  int rMbL[4][4];
  unsigned char avaimode;

  int tmpidx0;
  int tmpidx1;
#pragma HLS ARRAY_PARTITION variable=rMbL complete dim=1
#pragma HLS ARRAY_PARTITION variable=rMbL complete dim=2


#pragma HLS ARRAY_PARTITION variable=predL complete dim=1
#pragma HLS ARRAY_PARTITION variable=predL complete dim=2

  unsigned char inter_temp0[9][9];
  unsigned char inter_temp1[9][9];

#pragma HLS ARRAY_PARTITION variable=inter_temp0 complete dim=1
#pragma HLS ARRAY_PARTITION variable=inter_temp0 complete dim=2

#pragma HLS ARRAY_PARTITION variable=inter_temp1 complete dim=1
#pragma HLS ARRAY_PARTITION variable=inter_temp1 complete dim=2

#pragma HLS ARRAY_PARTITION variable=mvd0 complete dim=1
#pragma HLS ARRAY_PARTITION variable=mvd1 complete dim=1
```

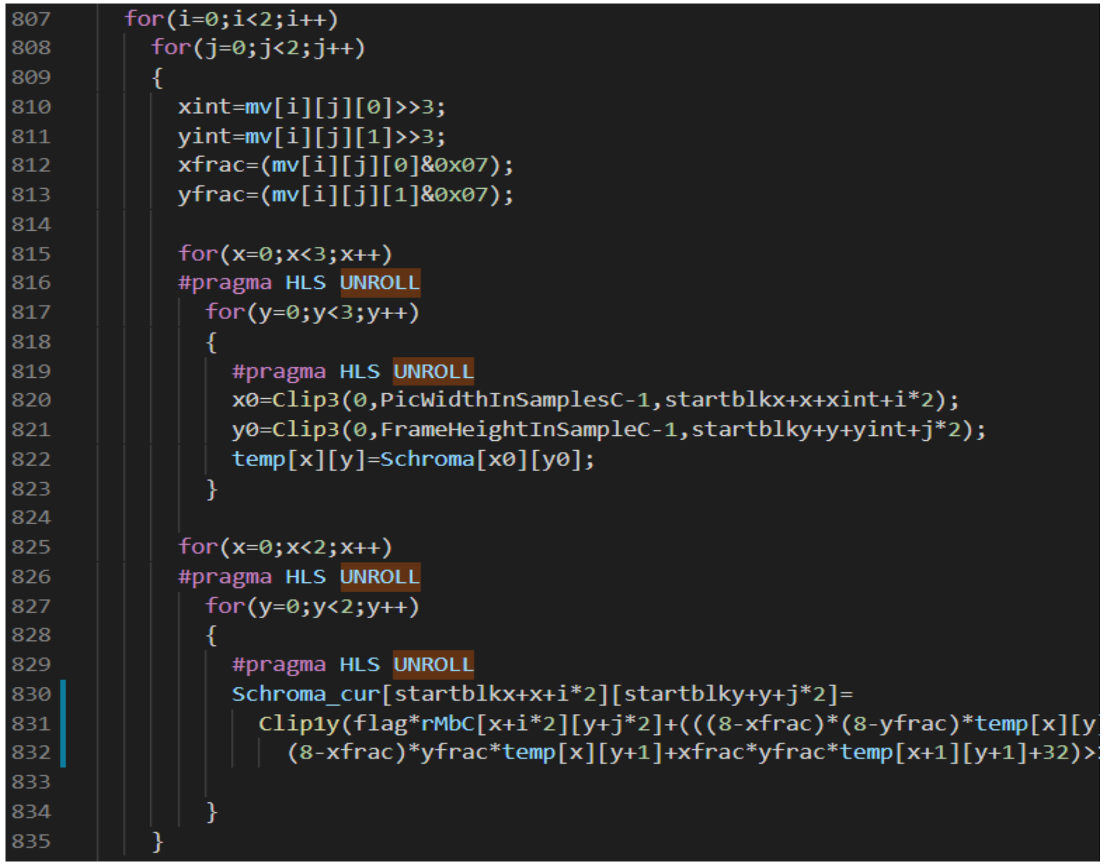
About this project

- Reference frames and NALU are stored in DDR
- Array partition into smaller BRAMs or FF



Challenges 1 – Memory bound Limits parallelism efficiency

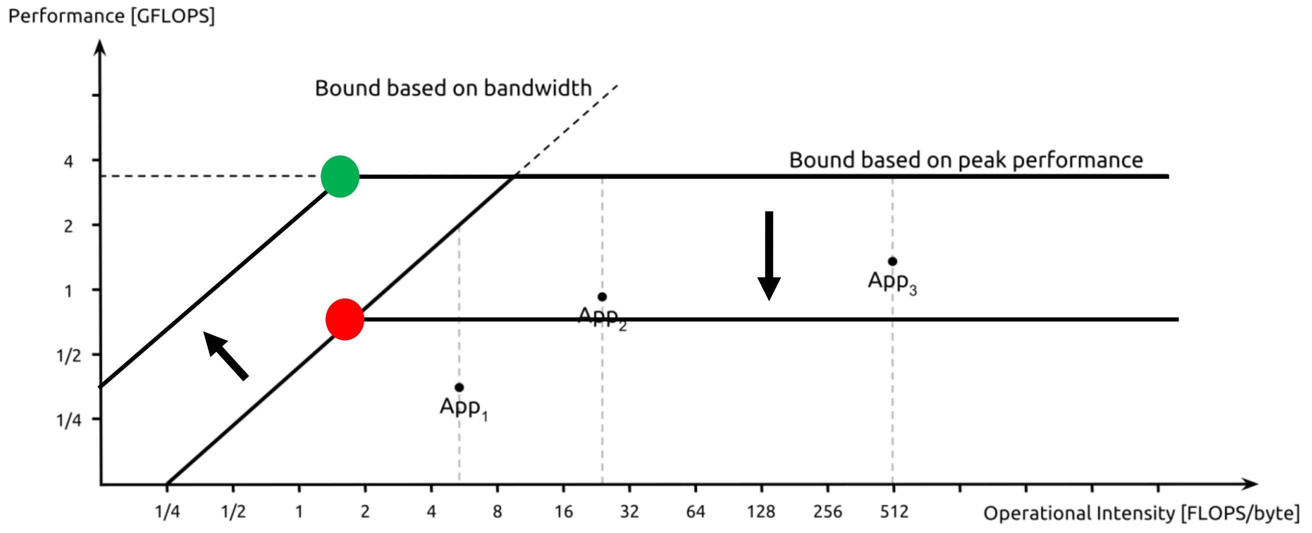
- When accessing data in DDR, the UNROLL pragma has no function if the UNROLL loop contains read/write operation on it
- Throughput is bounded by the low memory bandwidth



```
for(i=0;i<2;i++)
  for(j=0;j<2;j++)
  {
    xint=mv[i][j][0]>>3;
    yint=mv[i][j][1]>>3;
    xfrac=(mv[i][j][0]&0x07);
    yfrac=(mv[i][j][1]&0x07);

    for(x=0;x<3;x++)
    #pragma HLS UNROLL
      for(y=0;y<3;y++)
      {
        #pragma HLS UNROLL
        x0=Clip3(0,PicWidthInSamplesC-1,startblkx+x+xint+i*2);
        y0=Clip3(0,FrameHeightInSampleC-1,startblky+y+yint+j*2);
        temp[x][y]=Schroma[x0][y0];
      }

    for(x=0;x<2;x++)
    #pragma HLS UNROLL
      for(y=0;y<2;y++)
      {
        #pragma HLS UNROLL
        Schroma_cur[startblkx+x+i*2][startblky+y+j*2]=
          Clip1y(flag*rMbC[x+i*2][y+j*2]+(((8-xfrac)*(8-yfrac)*temp[x][y
            (8-xfrac)*yfrac*temp[x][y+1]+xfrac*yfrac*temp[x+1][y+1]+32)>

      }
  }
```

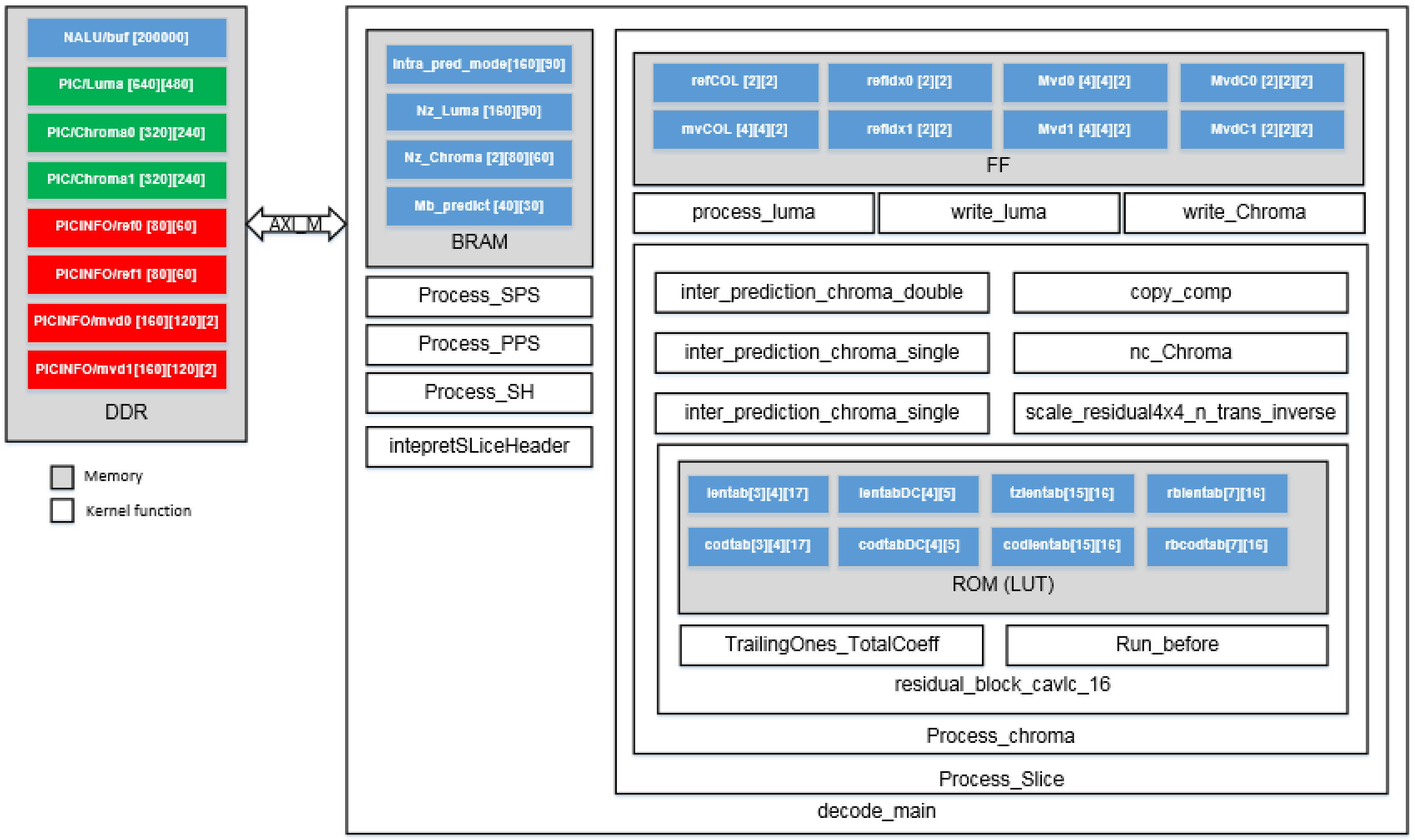
Solutions to Challenge 1

- To achieve higher utilization, we can
  - Increase bandwidth
  - Change UNROLL to PIPIELINE for all for loop that relates to memory access



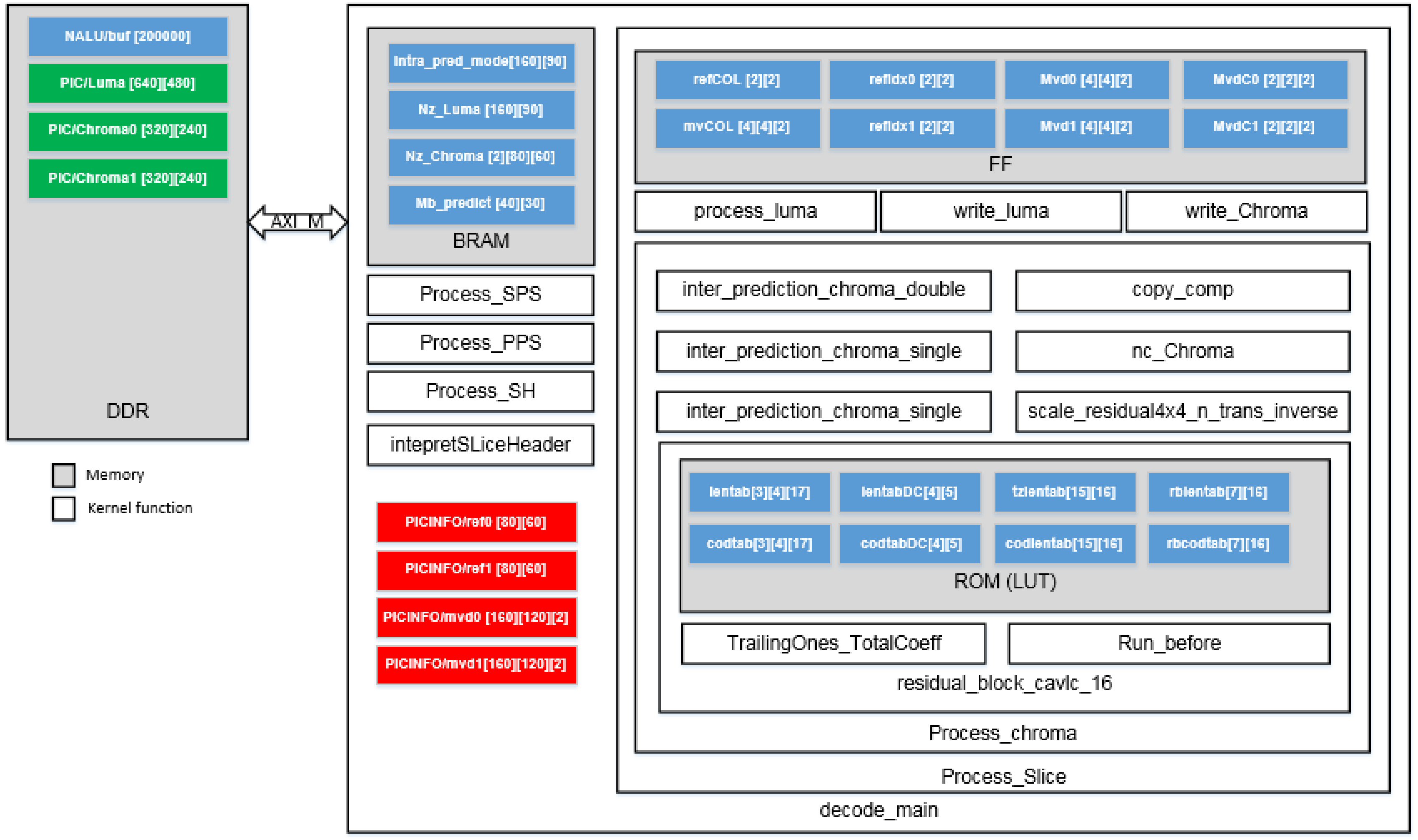
Challenges 2 – Long access time for data from DDR

- Access data from DDR may be slow and non-parallelizable



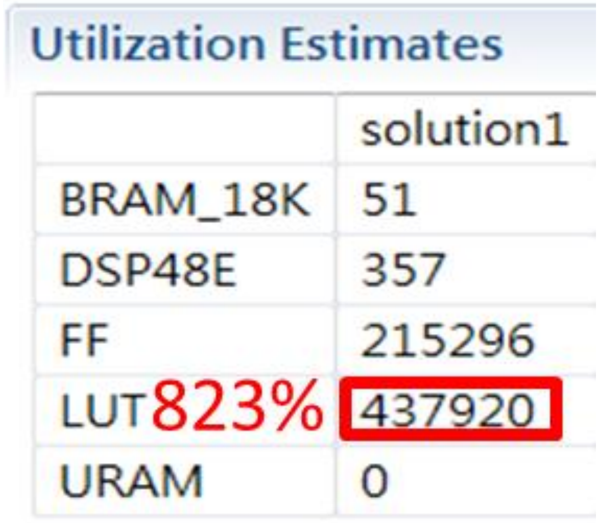
Solutions to Challenge 2

- Only PIC needs to be stored in DDR due to access of host program
- Array partition PIC_INFO to enhance parallelism



Challenges 3 – Large ROMs requires large LUT resource

- ROMs requires large LUT resource
- Implement ROM by BRAM
  - Transfer constant data through DDR before executing decode_main kernel

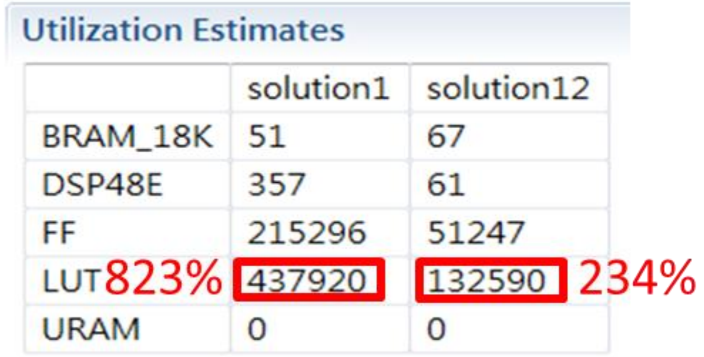


Utilization Estimates

	solution1
BRAM_18K	51
DSP48E	357
FF	215296
LUT 823%	437920
URAM	0

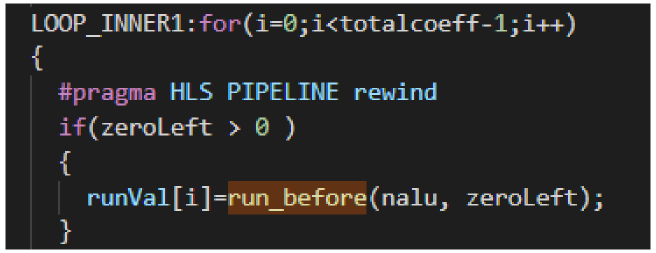
Insight

- The resource on Zedboard is not enough
- Reduce LUT
  - Array decoder logic is large (large LUT usage)
  - Trade time for area
    - Remove all UNROLL and ARRAY PARTITION pragmas
    - Add PIPELINE pragmas in inner-loops instead of outer
  - Change division/remainder of 2^n to shifter/and
  - Replace ROM to BRAM

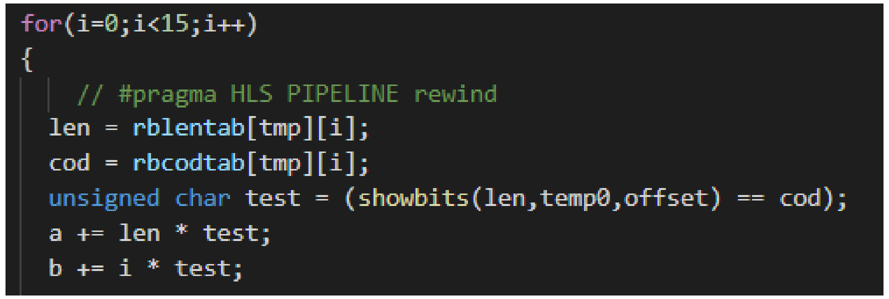


Utilization Estimates

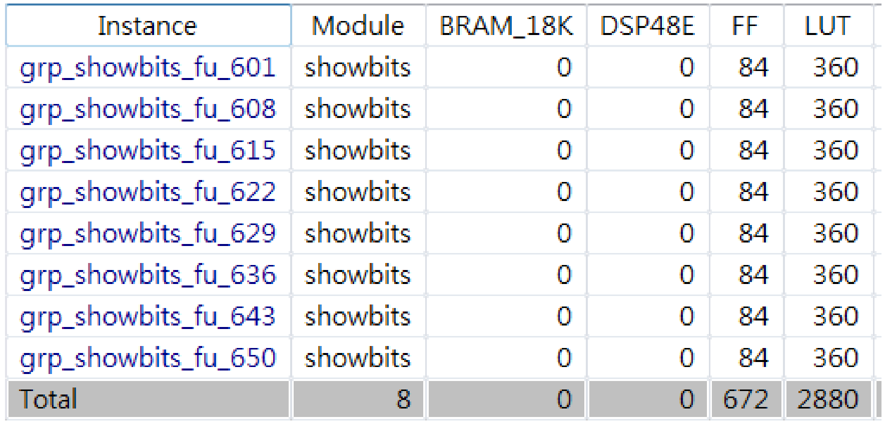
	solution1	solution12
BRAM_18K	51	67
DSP48E	357	61
FF	215296	51247
LUT 823%	437920	132590 234%
URAM	0	0



```
LOOP_INNER1:for(i=0;i<totalcoeff-1;i++)
{
  #pragma HLS PIPELINE rewind
  if(zeroLeft > 0 )
  {
    runVal[i]=run_before(nalu, zeroLeft);
  }
```



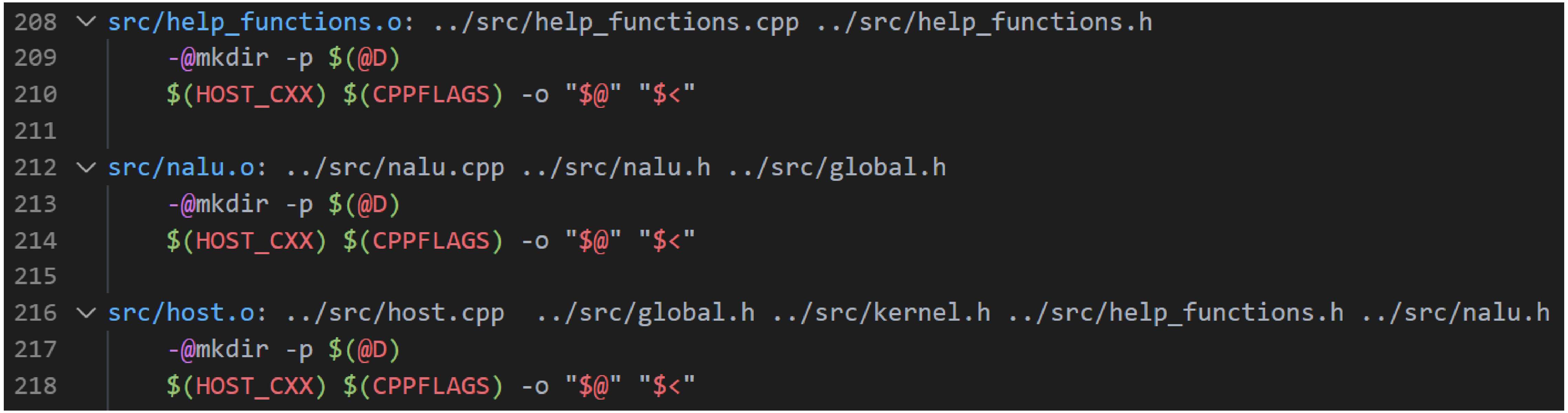
```
for(i=0;i<15;i++)
{
    // #pragma HLS PIPELINE rewind
  len = rblentab[tmp][i];
  cod = rbcodtab[tmp][i];
  unsigned char test = (showbits(len,temp0,offset) == cod);
  a += len * test;
  b += i * test;
```



Instance	Module	BRAM_18K	DSP48E	FF	LUT
grp_showbits_fu_601	showbits	0	0	84	360
grp_showbits_fu_608	showbits	0	0	84	360
grp_showbits_fu_615	showbits	0	0	84	360
grp_showbits_fu_622	showbits	0	0	84	360
grp_showbits_fu_629	showbits	0	0	84	360
grp_showbits_fu_636	showbits	0	0	84	360
grp_showbits_fu_643	showbits	0	0	84	360
grp_showbits_fu_650	showbits	0	0	84	360
Total	8	0	0	672	2880

Vitis Acceleration

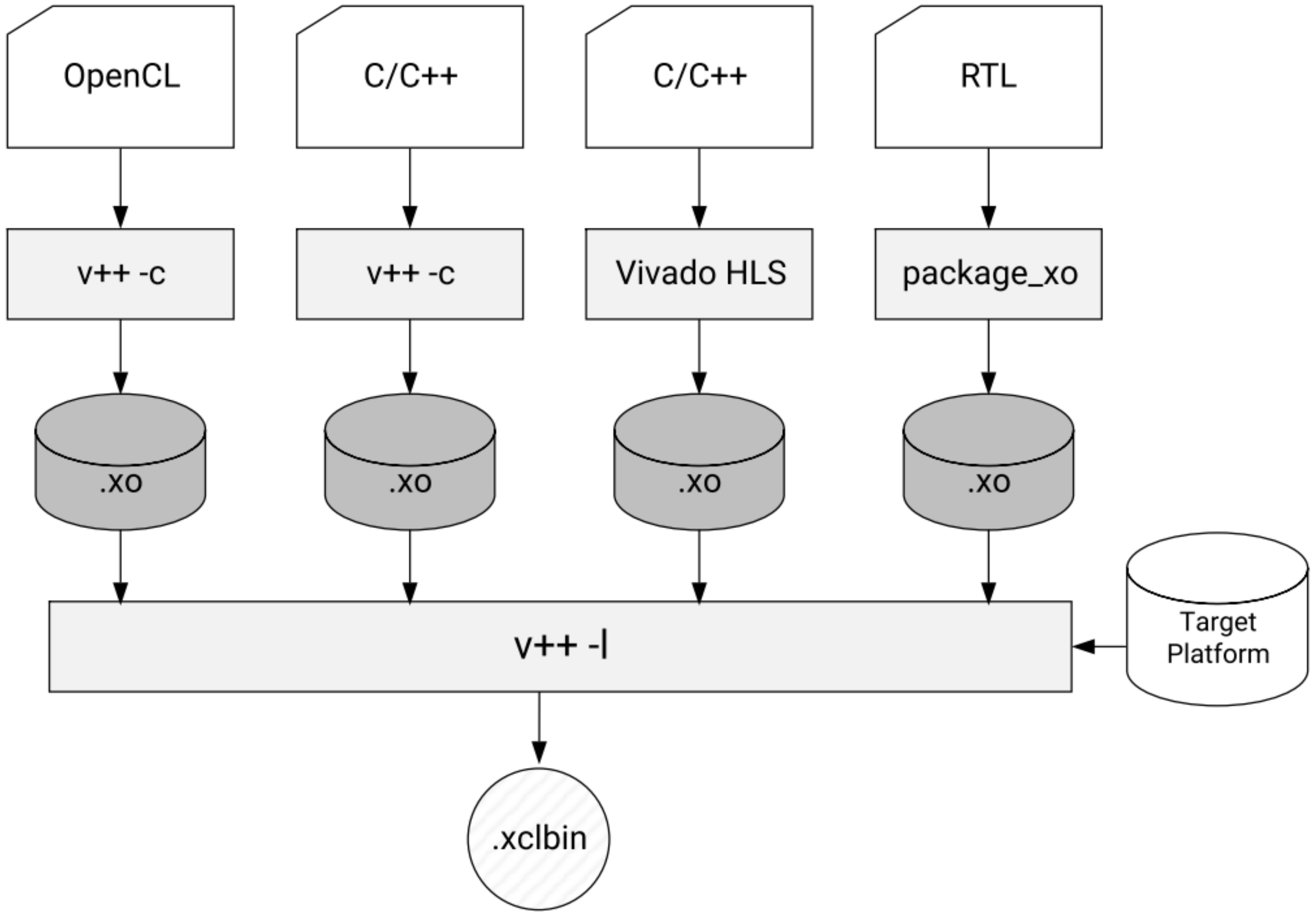
- Building the Host Program
  - Each source file of the host application is compiled into an object file (.o) using the g++ compiler



```
src/help_functions.o: ../src/help_functions.cpp ../src/help_functions.h
	-@mkdir -p $(@D)
	$(HOST_CXX) $(CPPFLAGS) -o "$@" "$<"

src/nalu.o: ../src/nalu.cpp ../src/nalu.h ../src/global.h
	-@mkdir -p $(@D)
	$(HOST_CXX) $(CPPFLAGS) -o "$@" "$<"

src/host.o: ../src/host.cpp  ../src/global.h ../src/kernel.h ../src/help_functions.h ../src/nalu.h
	-@mkdir -p $(@D)
	$(HOST_CXX) $(CPPFLAGS) -o "$@" "$<"
```

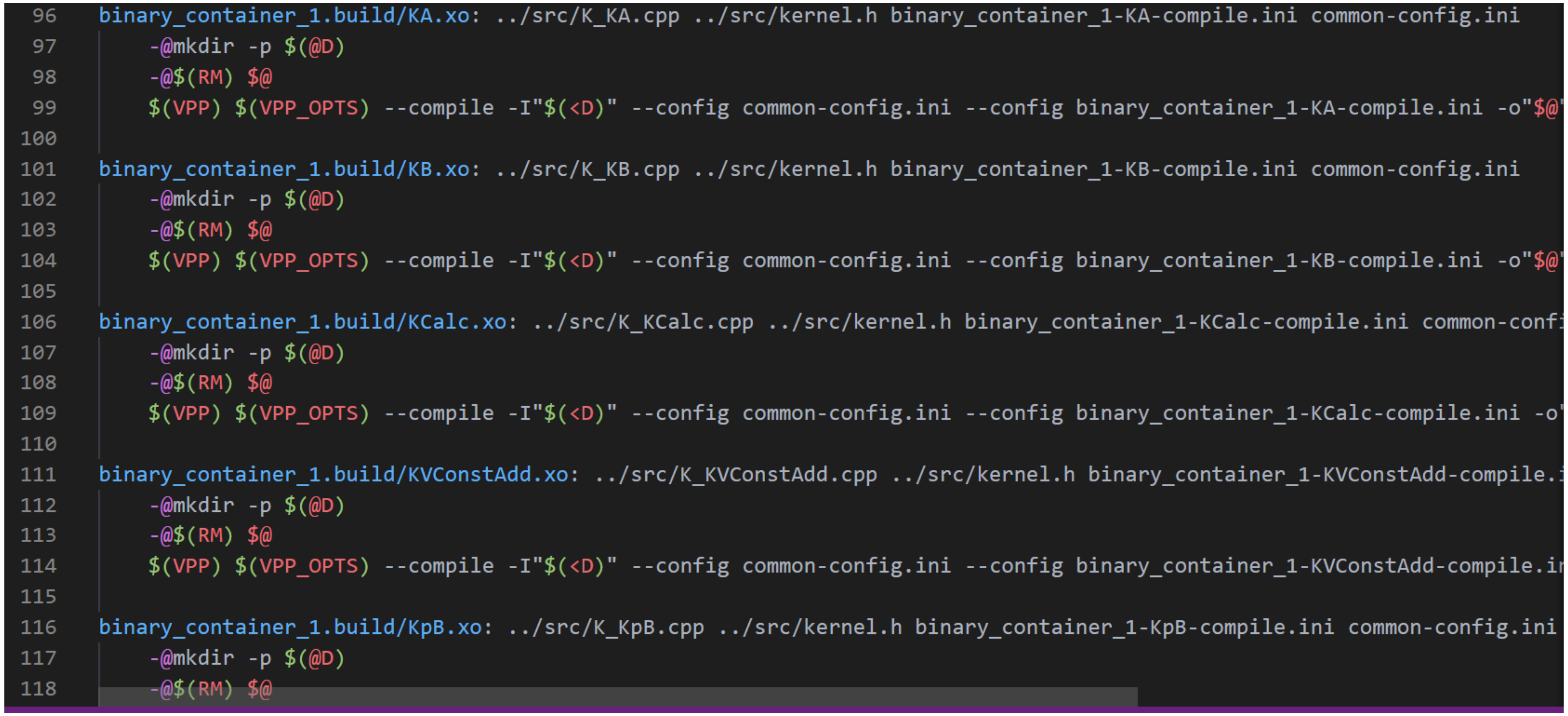
Vitis Acceleration

- Building the FPGA Binary



Makefile

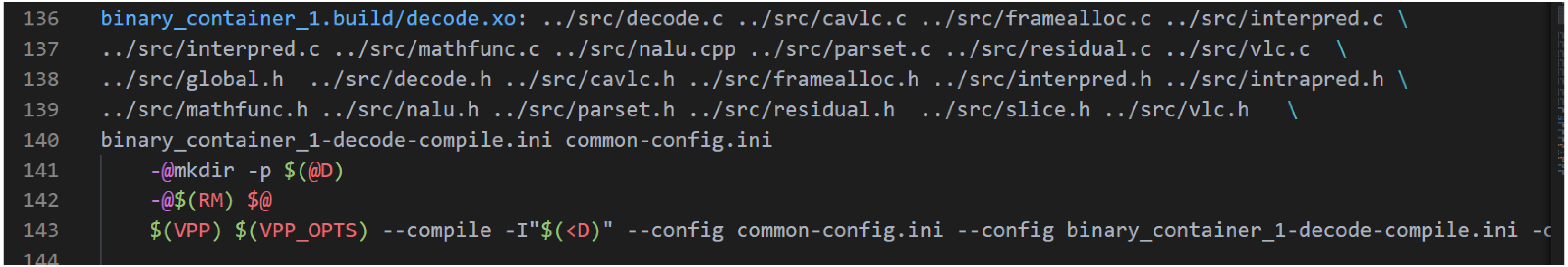
- For separate C function:
  – In Lab3, compiling every kernel code into different Xilinx object file (.xo)
  – Linking all .xo files into an FPGA binary file (.xclbin)



```
binary_container_1.build/KA.xo: ../src/K_KA.cpp ../src/kernel.h binary_container_1-KA-compile.ini common-config.ini
	-@mkdir -p $(@D)
	-@$(RM) $@
	$(VPP) $(VPP_OPTS) --compile -I"$(<D)" --config common-config.ini --config binary_container_1-KA-compile.ini -o"$@'

binary_container_1.build/KB.xo: ../src/K_KB.cpp ../src/kernel.h binary_container_1-KB-compile.ini common-config.ini
	-@mkdir -p $(@D)
	-@$(RM) $@
	$(VPP) $(VPP_OPTS) --compile -I"$(<D)" --config common-config.ini --config binary_container_1-KB-compile.ini -o"$@'

binary_container_1.build/KCalc.xo: ../src/K_KCalc.cpp ../src/kernel.h binary_container_1-KCalc-compile.ini common-confi
	-@mkdir -p $(@D)
	-@$(RM) $@
	$(VPP) $(VPP_OPTS) --compile -I"$(<D)" --config common-config.ini --config binary_container_1-KCalc-compile.ini -o'

binary_container_1.build/KVConstAdd.xo: ../src/K_KVConstAdd.cpp ../src/kernel.h binary_container_1-KVConstAdd-compile.i
	-@mkdir -p $(@D)
	-@$(RM) $@
	$(VPP) $(VPP_OPTS) --compile -I"$(<D)" --config common-config.ini --config binary_container_1-KVConstAdd-compile.ir

binary_container_1.build/KpB.xo: ../src/K_KpB.cpp ../src/kernel.h binary_container_1-KpB-compile.ini common-config.ini
	-@mkdir -p $(@D)
	-@$(RM) $@
```

Makefile

- For sub-function calls case:
  - Compiling all kernel code into ONE Xilinx object file (.xo), and let the kit add instance during Emulation by itself (top-down)
  - Linking all (.xo) files into an FPGA binary file (.xclbin)
  - You can also optimize a specific kernel function by package them into different (.xo) fil (bottom-up)



```
binary_container_1.build/decode.xo: ../src/decode.c ../src/cavlc.c ../src/framealloc.c ../src/interpred.c \
../src/interpred.c ../src/mathfunc.c ../src/nalu.cpp ../src/parset.c ../src/residual.c ../src/vlc.c  \
../src/global.h  ../src/decode.h ../src/cavlc.h ../src/framealloc.h ../src/interpred.h ../src/intrapred.h \
../src/mathfunc.h ../src/nalu.h ../src/parset.h ../src/residual.h  ../src/slice.h ../src/vlc.h   \
binary_container_1-decode-compile.ini common-config.ini
	-@mkdir -p $(@D)
	-@$(RM) $@
	$(VPP) $(VPP_OPTS) --compile -I"$(<D)" --config common-config.ini --config binary_container_1-decode-compile.ini -c
```

Future Work

- Keep working on this project
- U50 Implementation