EMAX7SPC-0001
Ver.0.1: Sep. 1 2023
Ver.0.2: Jan. 1 2024

EMAX7/ACAP (IMAX3) Architecture Handbook
– In-Memory Accelerator eXtension –

Nara Institute of Science and Technology
Computing Architecture Laboratory
Accelerator Group

Table of contents

List of Figures

List of Tables

Chapter 1

IMAX3 Hardware

1.1 Chiplet and Vector Length

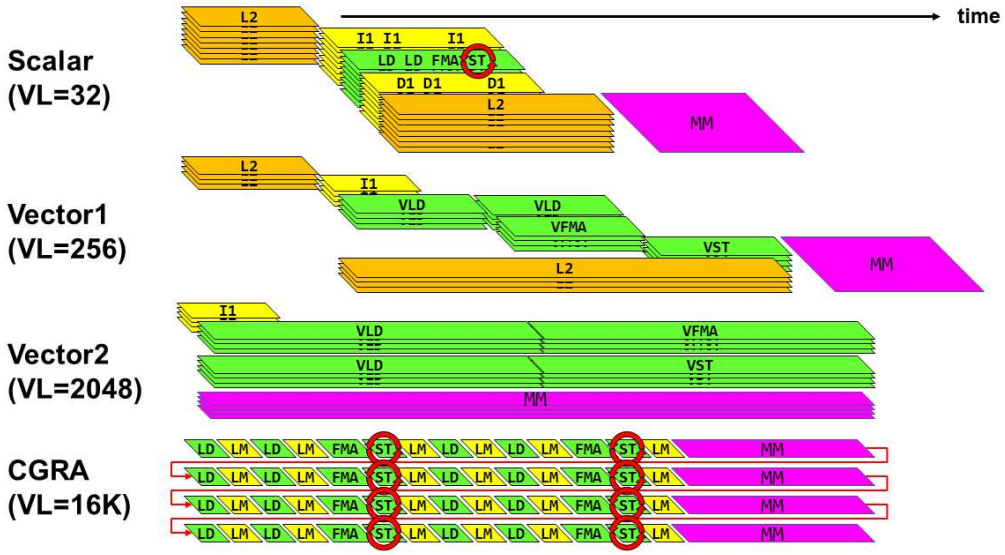


Figure.1.1: Variation of Vector Length

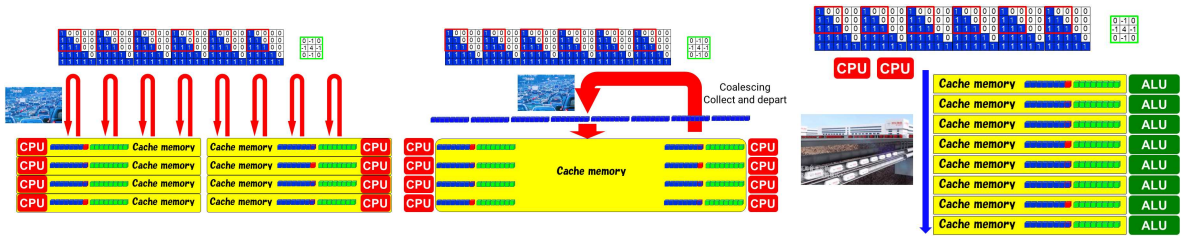


Figure.1.2: Multicore system, GPGPU and CGRA

Chiplets require sufficient vector length to minimize chip-to-chip data transfer overhead. And IMAX3 is a chiplet-oriented architecture. Let's look at the vector length in Fig.1.1. A scalar processor has SIMD for about 32 elements. SIMD for 256 elements or more is called a vector. Vector 1 is connected to the cache memory. Vector 2 is directly connected to the main memory, and the vector length is about 2048. The CGRA can have a sandwich structure of ALU and 64KB of cache memories. The vector length is now 16K. By encapsulating irregular access patterns in cache memory, the main memory can keep high speed with regular access patterns.

Figure1.2 illustrates typical execution models. Let's assume one car is one CPU. Blue data is the

input image, and green data is the weight. Every CPU tries to get the missing data in cache memory, even if it's the same, and we cannot estimate when the data will arrive. GPGPU also has many cores, but the cars are aligned and coalesced as much as possible before they depart. By merging the departure and arrival, we can reduce traffic congestion. However, if the destinations are scattered, GPGPU can do nothing. Whether coalescing is possible or not depends on the programming skill. The right side is IMAX. There are a few CPUs on top. A lot of cache memories do not go to get data and wait for the data provided by the CPU. The CPU can send the green weight data at once. We can reduce the amount of data transmission and energy.

1.2 Independent network for memory and execution

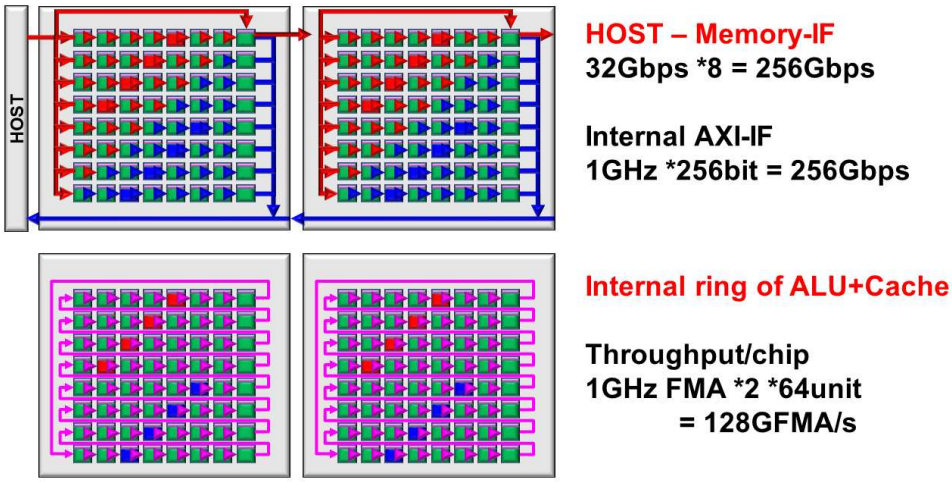


Figure.1.3: Execution network and Memory network

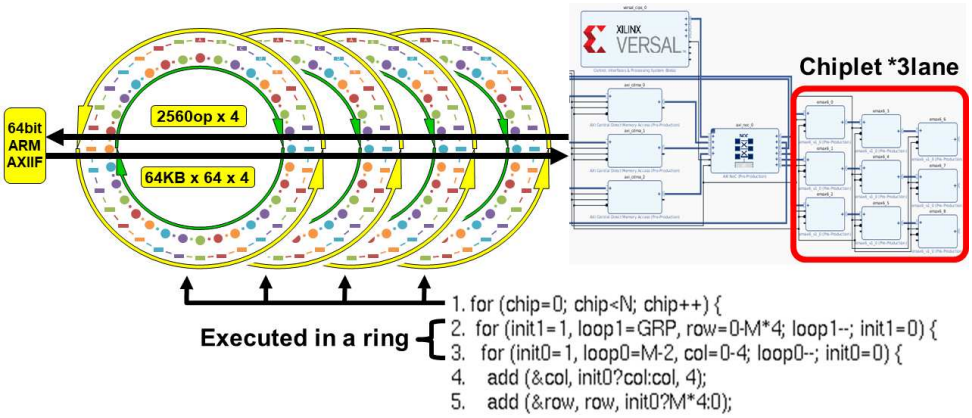


Figure.1.4: IMAX2 multichip structure

As shown in Fig.1.3, there are two types of networks in IMAX2. The latency is reduced by grouping eight units and connecting them in parallel to the memory interface. For computation, 64 units are connected in a ring within each chip, and the combination of computation and local memory can be changed like a combination lock. A ring structure is helpful for stencil computations. By sliding the mapped operations, the pair of ALU and cache memory can be adjusted. Much of the data in cache memory can be reused. To reduce the overhead, the triple loop can be mapped on IMAX as shown in Fig.1.4. The outermost loop is mapped on multiple chips, and the inner double loop is mapped on each chip. Fig.1.5 illustrates how four logical units (one physical unit) manage the triple loops. IMAX3 has multiple memory ports and multiple IMAX2 lanes, as shown in Fig.1.6.

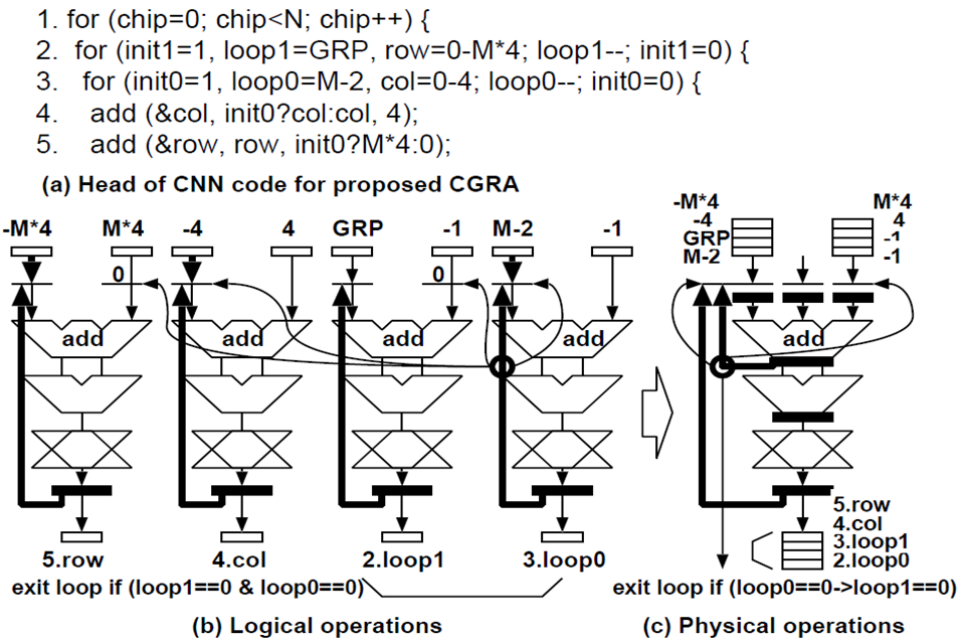


Figure.1.5: Burst execution of triple loops

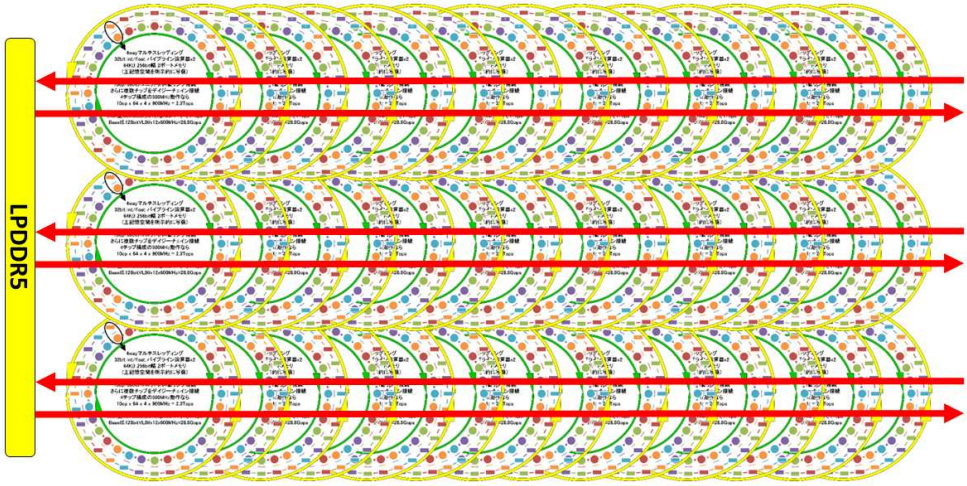


Figure.1.6: IMAX3 multilane structure

1.3 Multilevel pipelining

Figure1.7 illustrates the multiple lanes of IMAX2 connected to LPDDR5 memory, and the first level (micro) pipelining in each lane. The micro pipelining is the basic mode of CGRAs. Each lane may have multiple IMAX2 chips. This mode can follow the traditional and sequential program, and can reduce the compilation time. Figure1.8 illustrates the multiple lanes of IMAX2 and the second level (medium) pipelining in each lane. The medium pipelining is provided by the double buffering in cache memory blocks in each lane, and can follow the multiple stages in sorting, hash function, and FFT. Figure1.9 illustrates the macro pipelining of IMAX3. The multiple lanes are concatenated as a pipeline through LPDDR5. Each lane may have micro and medium pipelining. Figure1.10 shows the final goal of IMAX3. Each lane has multiple IMAX2 chips. Many lanes and CPUs are employed, and many kinds of kernels are mapped simultaneously as shown in Fig.1.11.

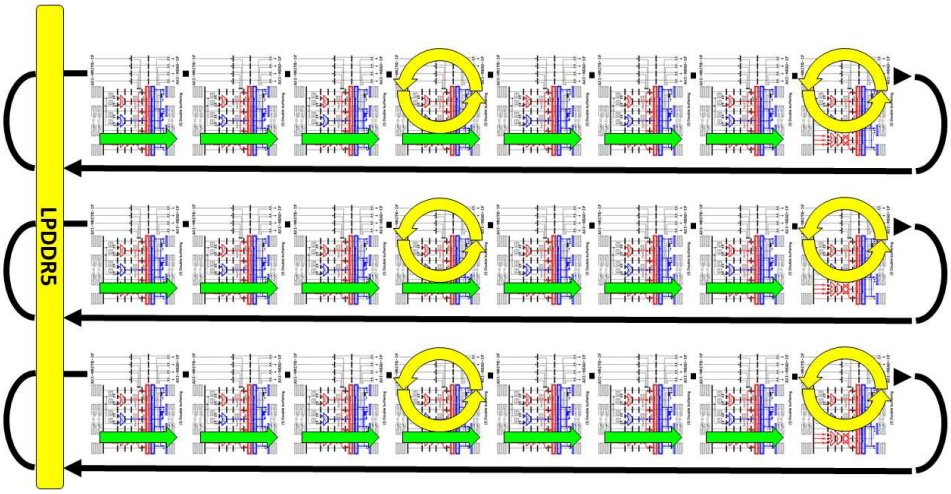


Figure.1.7: Micro pipelining

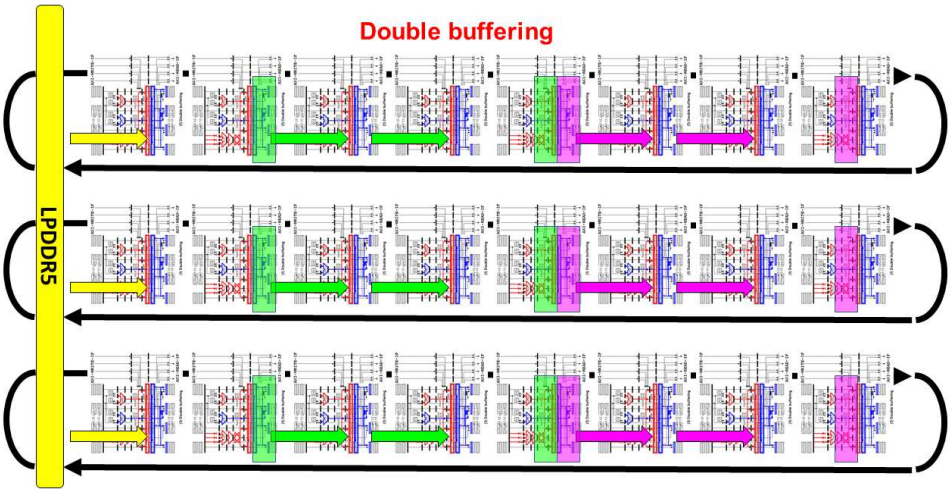


Figure.1.8: Medium pipelining

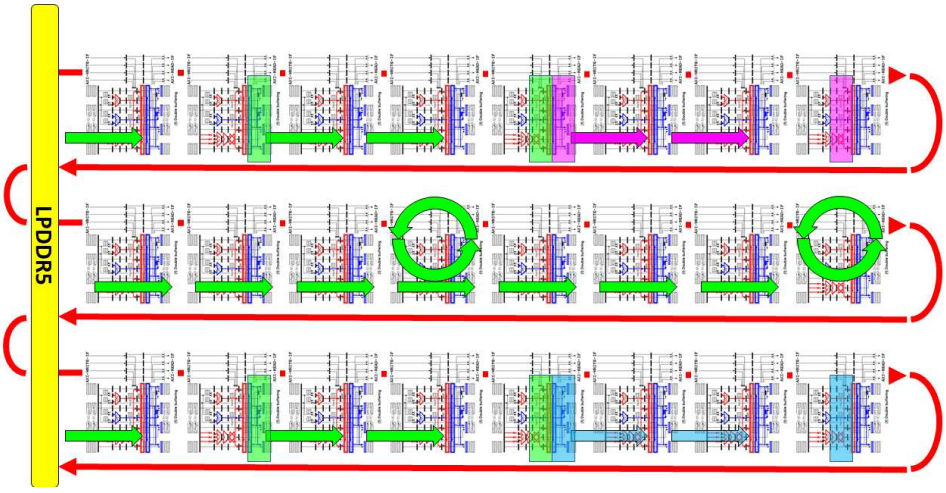


Figure.1.9: Macro pipelining

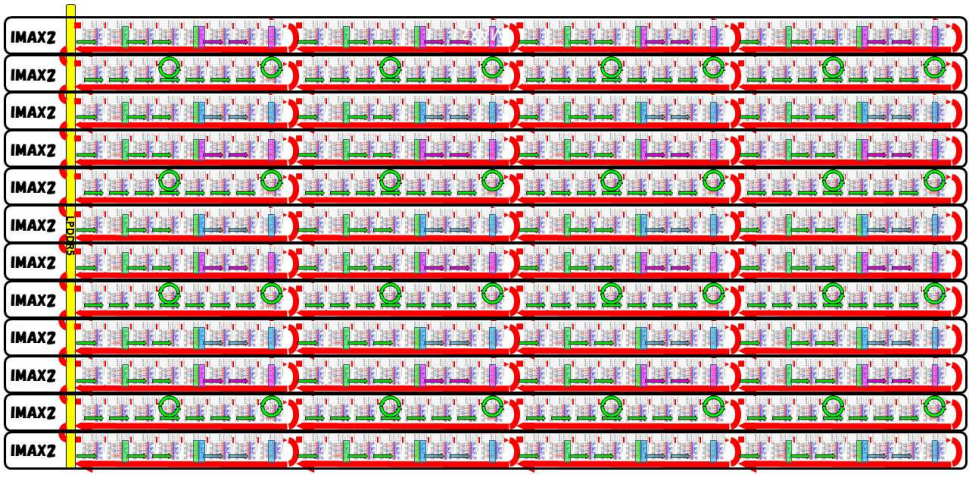


Figure.1.10: Multilevel pipelining

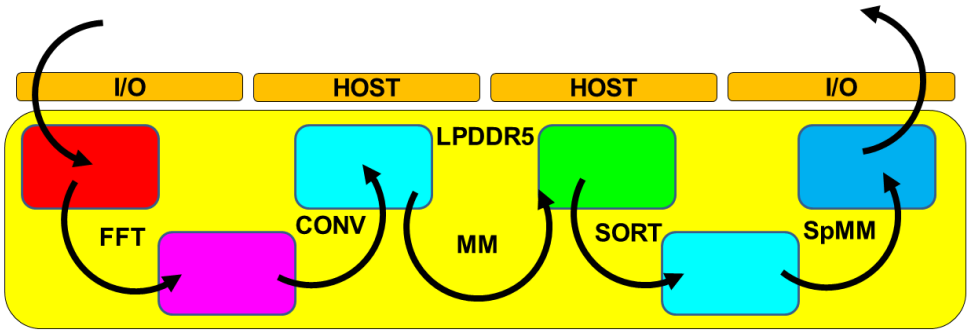


Figure.1.11: Mapping of application kernels

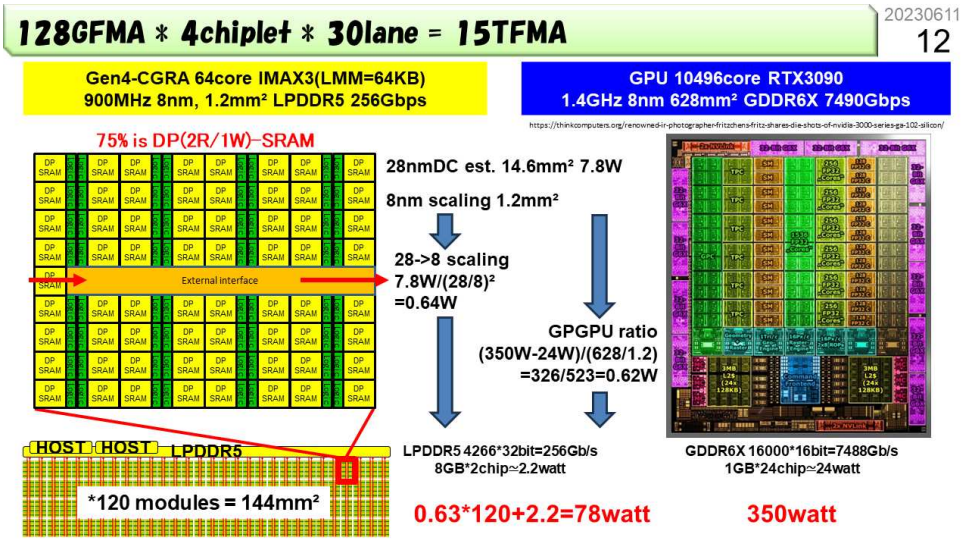


Figure.1.12: Area estimation

1.4 Prototype

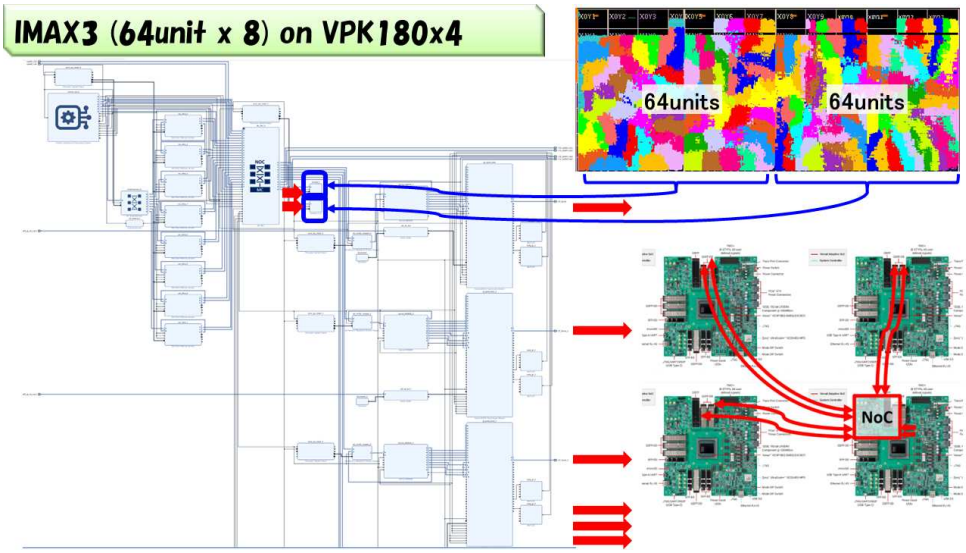


Figure.1.13: Prototype of IMAX3

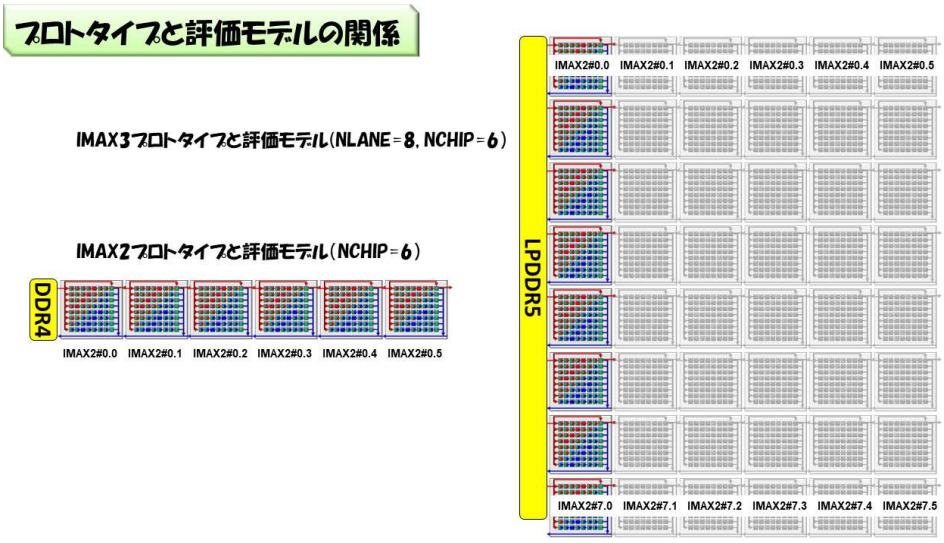


Figure.1.14: Evaluation models

As shown in Fig.1.12, 75% area of IMAX3 is memory blocks. If each port of the LPDDR5 has four modules of 64-unit IMAX, 10240 operations can be mapped. If 30 ports are available, 307200 operations can be mapped. If we can fabricate with 8nm technology, 120 modules of IMAX will occupy the 144mm square. Figure1.13 is the ongoing project to scale up the IMAX2 to IMAX3. VPK180 has two sets of IMAX2 and can connect eight IMAX2 through the NoC. Figure 1.14 is a performance evaluation model using a prototype. Eight units will be implemented: IMAX2#0.0, IMAX2#1.0, ..., IMAX2#7.0. Each IMAX2 corresponds to a configuration of NLANE=8 and NCHIP=1 in the IMAX2 application program. On the other hand, it is also possible to run the IMAX2 application program by writing NLANE=8 and NCHIP=6. This method corresponds to a configuration with multiple IMAX2 lanes in a cascade configuration. However, since only the first IMAX2#*.0 is implemented, the execution result of subsequent IMAXs connected in cascade will be 0, and the overhead associated with cascade connection will not be measured. The execution speed is just a ideal value.

Chapter 2

IMAX3 Software

2.1 IMAX3 interface mapped on CPU memory space

IMAX3 is just a group of multiple lanes of IMAX2 connected to a large-capacity external memory. The hardware/software interface of IMAX3 is a set of control interfaces for each IMAX2.

2.2 Dataflow example

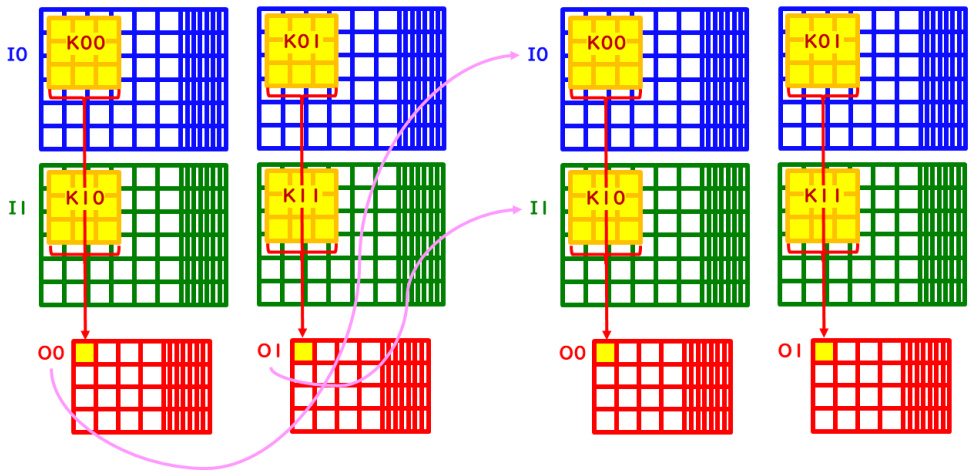


Figure.2.1: Tycal series of 4D-array convolution

Figure 2.1 shows a typical convolution operation. Multiplication of a part of the input data (I) and the corresponding kernel (K), and accumulation produce the result. The shape that 6x6 two-dimensional I0 overlap in the depth direction corresponds to multiple 6x6 images being included in one batch. I1 has a similar structure. For example, I0 corresponds to the blue component and I1 corresponds to the green component. There are many pairs of input data and kernels, and the vertical summation is the output O0. Since the input data is two-dimensional and the calculation is repeated by shifting the kernel K in the X and Y directions, the output O0 is also two-dimensional. Also, if you keep the input data (I) as is and replace only the kernel (K) and perform the same calculation, you will find another output O1. In other words, the input data (I), kernel (K), and output data (O) are all 4-dimensional arrays. Furthermore, in the multilayer convolution operation, the same calculation is repeated using the output data (O) as the next input data (I).

There are several ways to perform the above convolution operations using IMAX3. Fig.2.2 is the basic form of two-dimensional convolution. Each IMAX unit contains cache memory and a computing unit that are controlled by the host driver. Blue input data in main memory is broadcast to the blue part of

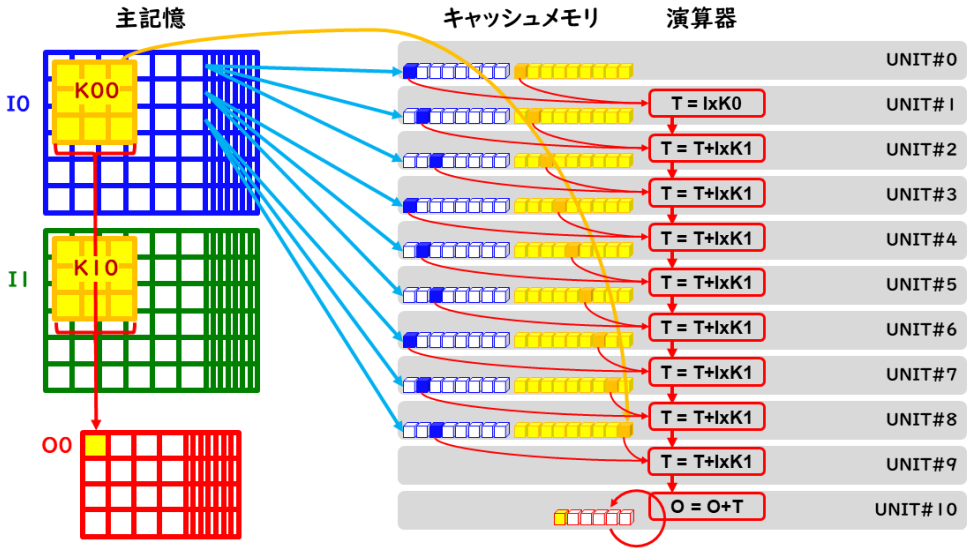


Figure.2.2: Typical mappting of convolution

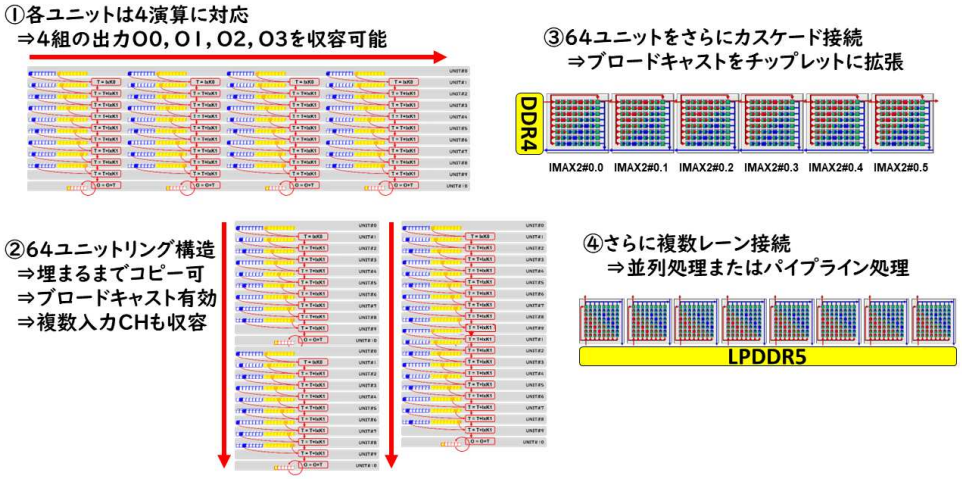


Figure.2.3: Several options to speedup convolution

the cache memory, and yellow kernels in main memory are broadcast to the yellow part. Unit 0 extracts the data corresponding to the upper left corner of the kernel and sends it to unit 1. Unit 1 performs the multiplication and sends the result to unit 2. By connecting the above, finally, unit 10 accumulates the sum of the 9 multiplication results to the red output, and completes one 3x3 convolution operation. All units perform calculations one after another while shifting the input data to the right, and the output data is updated continuously. The above is a typical CGRA calculation method.

A feature of IMAX is the large number of degrees of freedom in combining the above basic shapes according to the size of the 4-dimensional array (Fig.2.3). The goal of optimization is to reduce execution time, and the methods are broadcasting and maximum reuse of cache memory. One chip of IMAX2 can continuously execute double loops at once, and can map four sets of convolution operations into a logical four-column structure. What remains is which axis of the 4-dimensional data should be mapped to the double loop. Once you decide which axis of input I to select, the others will be determined automatically, so first select the axis of blue input data (I). There are four candidates: the X axis, Y axis, batch axis, and channel axis, as shown in Fig.2.4. However, the X-axis should be selected because the addresses are continuous. Similarly, the channel axis is efficient if used to fill 64 units, so the remaining axes have two choices: the Y axis or the batch axis.

Figure2.5 shows two mapping methods. When using the X and Y axes, the yellow data does not need

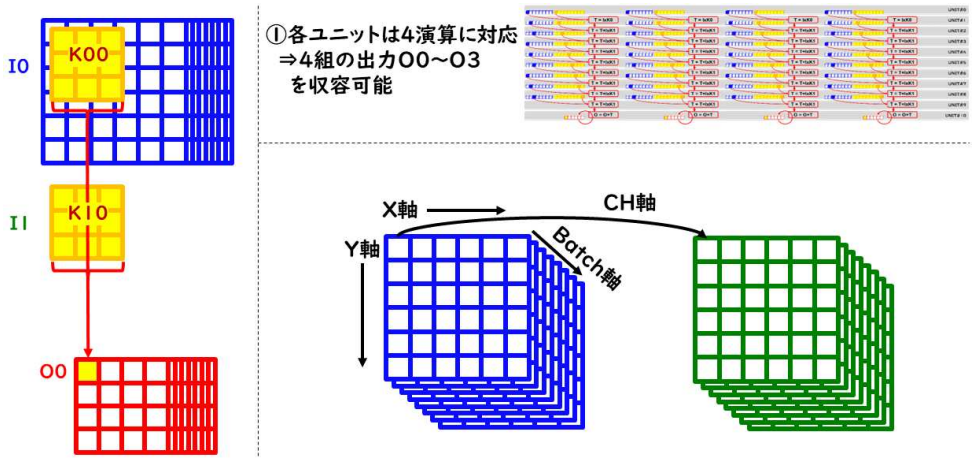


Figure.2.4: Selecting two axis from 4D array

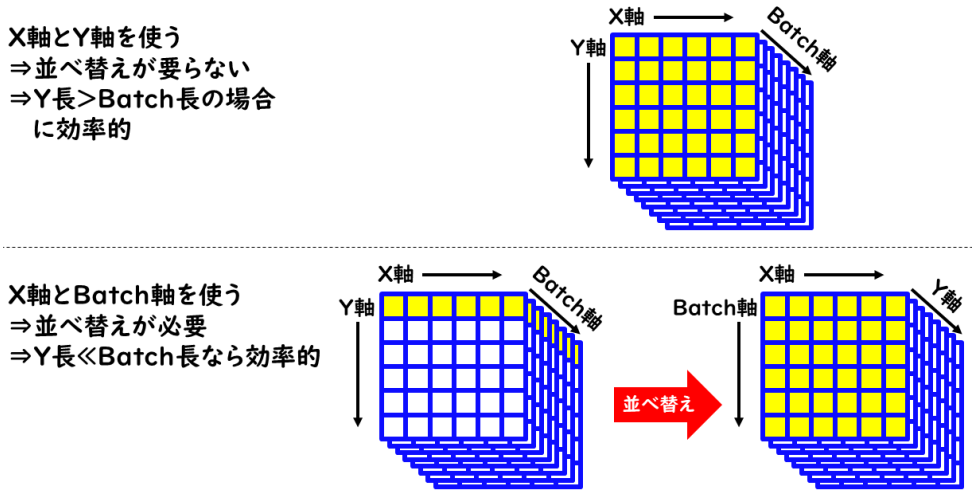
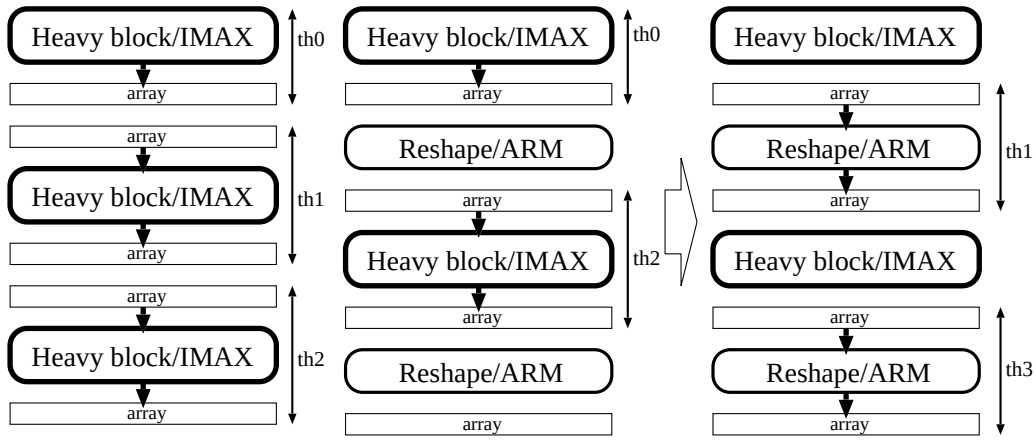


Figure.2.5: Two options to increase the length of the burst execution

to be reshaped because it is a continuous address. It is most efficient if the data size of one sheet, which is the product of the length of X and the length of Y, is stored in the cache memory within the unit. For example, if the cache memory is 16Kwords, it can accommodate up to 128x128. In the case of 256x256, you can keep this format and divide it into four in the Y direction. On the other hand, after multilayer convolution, the result is a small square such as 2x2, which shortens IMAX's continuous execution time and increases startup overhead. In such cases, use the batch axis instead of the Y axis. Although the overhead associated with reshaping increases, if the number of batches is 100, it becomes 2x100, which allows the data size to be increased and the startup overhead to be reduced. In the case of 3D convolution, you can similarly increase the continuous operation time of IMAX and reduce the number of startups.

2.3 Programming model of Macro pipelining

Figure2.6 is the programming model related to macro pipelining. In (a), there are three sections with long processing times, and the macro pipelining is performed. The main thread starts three threads and speeds up each section using IMAX. A double buffer is required for the data structure between threads so that the input and output of each thread do not interfere. On the other hand, (b) is a case where the long processing time is executed by IMAX, but there is short-term processing by the host, such as exchanging array indexes, between the heavy processing. By using short-time processing as a double buffer, memory usage can be reduced compared to applying double buffers between all threads.



(a) Contiguous heavy blocks (b) Combination of heavy and light blocks

Figure.2.6: Programming model

2.4 Programming style of Macro pipelining

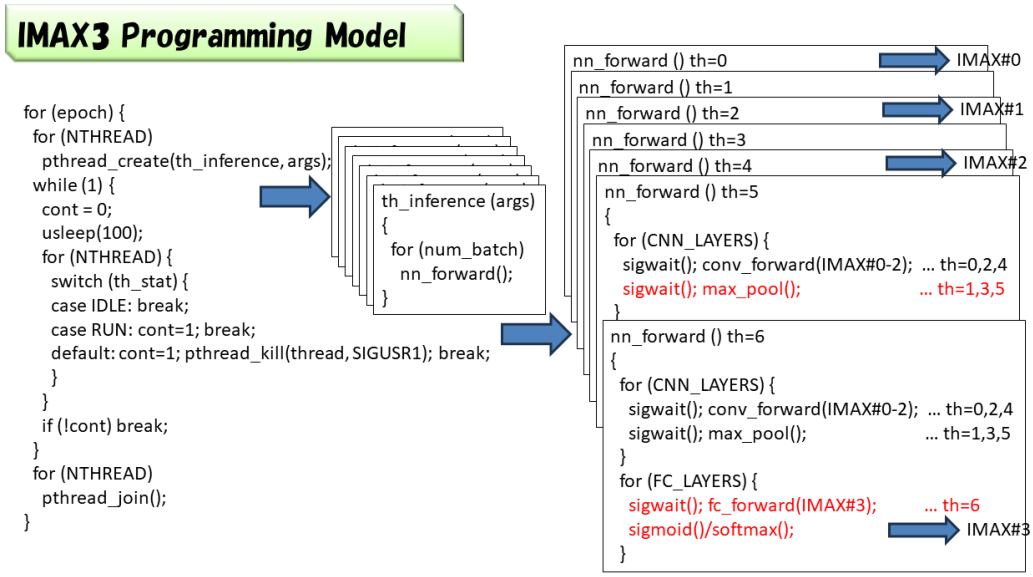


Figure.2.7: Programming style

Figure2.7 is the programming style. For the macro pipelining, multiple threads are started, and the function (th_inference) containing the loop structure is executed in parallel. The th_inference() repeatedly executes nn_forward(), which simulates a multilayer CNN. Although nn_forward() is executed simultaneously by multiple threads, each thread executes only the part it is responsible for, and performs pipeline processing while waiting to complete processing in the previous and subsequent layers. In each layer, function calls that take LANE as an argument use IMAX2. Multiple HOSTs and IMAX2 participate in macro pipelining. By using sigwait() instead of a busy loops, waiting between threads can be performed on HOST even if HOST does not have enough cores.

Chapter 3

Examples

3.1 Image Recognition (tsim)

MNIST

```
cent% make -f Makefile-cent.emax7nc all clean
cent% cd ../; tsim/tsim-cent.emax7nc -x -t -I0 -C1 -F1
```

```
acap% make -f Makefile-acap.emax7+dma all clean
acap% cd ../; tsim/tsim-acap.emax7+dma -x -t -I0 -C1 -F1
```

CIFAR10

```
cent% make -f Makefile-cent.emax7nc all clean
cent% cd ../; tsim/tsim-cent.emax7nc -x -t -I1 -C6 -F2
```

```
acap% make -f Makefile-acap.emax7+dma all clean
acap% cd ../; tsim/tsim-acap.emax7+dma -x -t -I1 -C6 -F2
```

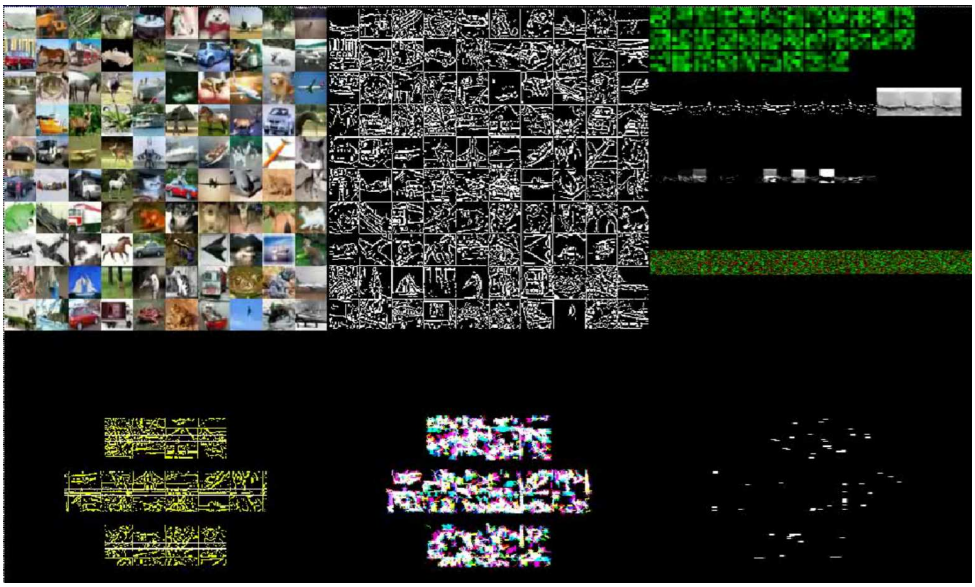


Figure.3.1: Image recognition (training + inference)

3.1.1 Header

IMAX3 uses ARM's multithreading to control multiple IMAX2 lanes. NTHREAD is the number of threads to be activated in ARM when multiple IMAX2 lanes are activated. EMAX_LANE is the upper limit of the control variable set to control the IMAX2 lanes activated simultaneously. Also, NLANE is the number of IMAX2 lanes detected in the actual machine. IMAX2 detected in the real machine over EMAX_LANE is not used. In other words, the value set in NLANE is always less than or equal to EMAX_LANE. Note that NTHREAD must be greater than or equal to NLANE.

```
#define MAX_NTHREAD 16
volatile struct th_inference_args {
  int      thid;
  int      stat;     /* 0:idle, 1:run, 2:wait (enq/deq, DMA, EXEC) */
  sigset_t sigset;   /* sys/_sigset.h 2B/4B */
  int      deq;
  int      enq;
  float4D *slice;
  CNNet   *net;
  int      batch_size;
  int      nchan;
  int      insize;
} th_inference_args[MAX_NTHREAD];
volatile int th_inference_retv[MAX_NTHREAD];
pthread_t    th_inference_t[MAX_NTHREAD];
void         th_inference(struct th_inference_args *);
```

3.1.2 IMAX3 thread driver

When launching multiple IMAX2 and configuring a macro pipeline, prepare a function wrapper (th_inference in the example below) that includes the IMAX2 kernel, and use ARM's pthread_create to launch NTHREAD threads. Thread arguments include thread number and enq/deq for pipeline synchronization.

```
int THREAD;
for (THREAD=0; THREAD<NTHREAD; THREAD++) {
  th_inference_args[THREAD].thid       = THREAD;
  th_inference_args[THREAD].stat       = 1; /* run */
  sigemptyset(&th_inference_args[THREAD].sigset);
  sigaddset(&th_inference_args[THREAD].sigset, SIGUSR1);
  pthread_sigmask(SIG_BLOCK, &th_inference_args[THREAD].sigset, NULL);
  th_inference_args[THREAD].enq        = 0;
  th_inference_args[THREAD].deq        = 0;
  th_inference_args[THREAD].slice      = &slice;
  th_inference_args[THREAD].net        = net;
  th_inference_args[THREAD].batch_size = batch_size;
  th_inference_args[THREAD].nchan      = nchan;
  th_inference_args[THREAD].insize     = insize;
}
if (NTHREAD > 1) {
  for (THREAD=0; THREAD<NTHREAD; THREAD++)
    pthread_create(&th_inference_t[THREAD], NULL, (void*)th_inference, &th_inference_args[THREAD]); /* 0-(NTHREAD-1) */
  while (1) {
    int cont = 0;
    usleep(4000); /* 4msec 2024/01/24 Nakashima */
    for (THREAD=0; THREAD<NTHREAD; THREAD++) {
      switch (th_inference_args[THREAD].stat) {
      case 0:            break;/* idle */
      case 1:  cont = 1; break;/* run */
      default: cont = 1; pthread_kill(th_inference_t[THREAD], SIGUSR1); break; /* wait */
      }
    }
    if (!cont) break;
  }
  for (THREAD=0; THREAD<NTHREAD; THREAD++)
    pthread_join(th_inference_t[THREAD], NULL);
}
else {
  for (THREAD=0; THREAD<NTHREAD; THREAD++)
    th_inference(&th_inference_args[THREAD]);
}
```

3.1.3 IMAX3 thread wrapper

Multiple wrappers are started at the same time. In order for multiple threads to work in the pipeline internally, the code should be wrriten to execute the part in charge based on the thread number given as an argument.

```
/* IMAX3 MACROPIPELIING for EVALUATION */
void th_inference(struct th_inference_args *args)
{
  int THREAD     = args->thid;
  float4D *slice = args->slice;
  CNNet *net     = args->net;
  int batch_size = args->batch_size;
  int nchan      = args->nchan;
  int insize     = args->insize;
  int j, k;
  /***********************************/
  /* TARGET of MACRO-PIPELINING/IMAX3 */
  /***********************************/
  slice->nstrides = batch_size;
  slice->nchannel = xtest.nchannel;
  slice->kstrides = xtest.kstrides;
  slice->stride_size = xtest.stride_size;
  for (j=0; j+batch_size<=xtest.nstrides; j+=batch_size) {
    if (THREAD == 0) {
      slice->data = &(xtest.data[j*xtest.stride_size*xtest.kstrides*xtest.nchannel]);
      if (cnn_mode) {
        if (enable_x11) {
          F4i2Ipl(batch_size, nchan, insize, insize, I, slice); /* 100batch x 28x28 x 1chan */
          copy_I_to_BGR(D, batch_size, insize, insize, I);
          BGR_to_X(0, D);
        }
        // copy data to input layer
        copy4D(&(net->ninput), slice);
        if (attn_mode) {
          attention(&(net->ninput), &(net->work), &(net->attention)); /* batch,RGB,H,W */
          if (enable_x11) {
            F4i2Ipl(batch_size, nchan, insize, insize, I, &(net->work)); /* 100batch x 28x28 x 1chan */
            copy_I_to_BGR(D, batch_size, insize, insize, I);
            BGR_to_X(3, D);
            F4i2Ipl(batch_size, nchan, insize, insize, I, &(net->ninput)); /* 100batch x 28x28 x 1chan */
            copy_I_to_BGR(D, batch_size, insize, insize, I);
            BGR_to_X(1, D);
          }
        }
      }
      if (eye_mode) {
        F4i2Ipl(batch_size, nchan, insize, insize, I, slice); /* 100batch x 28x28 x 1chan */
        copy_I_to_BGR(R, batch_size, insize, insize, I); /* R <- I */
        copy_I_to_BGR(L, batch_size, insize, insize, I); /* L <- I */
        if (enable_x11)
          BGR_to_X(0, R);
        /* pre-processing by eye-model */
        eyemodel(enable_x11, slit_type); /* L+R -> Sl+Sr */
        /* import Sr to hidden_layer */
        Ipl2F4h(10, WD, HT, Sr, Cr, R, &net->nhidden[0]); /* 100batch x 24x24 x 9chan -> hidden */
      }
    }
    nn_forward(/*MACROPIPE*/1, THREAD, net, c[input_type], f[input_type], &pred, spike_mode);
    if ((NTHREAD==1 || eye_mode)
      ||(CNN_DEPTH==1 && THREAD==2)
      ||(CNN_DEPTH==3 && THREAD==6)
      ||(CNN_DEPTH==4 && THREAD==8)
      ||(CNN_DEPTH==6 && THREAD==12)) {

      for (k=0;k<batch_size;k++) {
        float *A = &(pred.data[k*pred.stride_size]);
        nerr += (MaxIndex(A, pred.stride_size) != ytest[j+k]);
      }
      if (enable_x11) {
        clear_BGR(D);
        copy_H_to_BGR(D+WD*(HT*1/4), &net->nhidden[0]);
        copy_H_to_BGR(D+WD*(HT*2/4), &net->nhidden[CNN_DEPTH-1]);
        BGR_to_X(2, D);
        while (x11_checkevent());
      }
    }
  }
  /*********************************/
  /* END of MACRO-PIPELINING/IMAX3 */
  /*********************************/
  args->stat = 0; /* idle */
}
```

3.1.4 IMAX3 region

The nn_forward below is the top-level function that performs the inference. Currently, the argument THREAD is manually associated with the IMAX2 lane number (LANE) to be used. In addition, enq/deq is used to synchronize the macro pipeline. In the future, when automation tools are completed, manual work will be unnecessary.

```
void nn_forward(int MACROPIPE, int THREAD, CNNet *net, struct c *c, struct f *f, float2D *oubatch, int spike_mode)
{ /* NTHREAD 1:MACRO_PIPE_OFF 2-:MACRO_PIPE_ON */
  /*                              EMAX7:NTHREAD=8 other:NTHREAD=1 */
  /*                               ZYNQ:NLANE=X                   */
  /*                               othr:NLANE=4                   */
  /* train:     nn_forward(0, 1)                                  */
  /* inference: nn_forward(1, T)                                  */
  /* camera:    nn_forward(0, 1)                                  */
  /* th#>0 の場合, 前段 enq==deq なら待機 最終 th#未満の場合, 自段 enq!=deq なら待機 */
  /* 待機状態になければ, 前段 deq=1-deq に更新                                 */
  /* 自段 enq で自身の dbuf 選択                                          */
  /* 最後に,            自段 enq=1-enq に更新                             */
  /* DBUF の場合                                                          */
  /* th#0      ****0* ****1* ****2*  CNN0:ninput->nhidden[0]              */
  /*      enq 0     11     00     11                                      */
  /*      deq 0     01     10     01                                      */
  /* th#1     -------*0*    *1*    *2*  nhidden[0]->npool[0]              */
  /*      enq 0        11     00     11                                   */
  /*      deq 0        01     10     01                                   */
  /* th#2     ----------****0* ****1* ****2*  CNN1:npool[0]->nhidden[1]   */
  /*      enq 0              11     00     11                             */
  /*      deq 0              01     10     01                             */
  /* th#3     ----------------*0*    *1*    *2*  nhidden[1]->npool[1]     */

  /* NHIDDEN/NPOOL を DBUF として使う場合                                  */
  /* th#0      ****0*  ****1*  ****2*  CNN0:ninput->nhidden[0]            */
  /*      enq 0     1 1     0 0     1 1                                   */
  /*      deq 0     0 1     1 0     0 1                                   */
  /* th#1     ------*0*     *1*     *2*  nhidden[0]->npool[0]            */
  /*      enq 0       1     1 0     0 1                                   */
  /*      deq 0       0     1 1     0 0     1                             */
  /* th#2     ----------****0*  ****1*  ****2*  CNN1:npool[0]->nhidden[1] */
  /*      enq 0             1 1     0 0     1 1                           */
  /*      deq 0             0 1     1 0     0 1                           */
  /* th#3     --------------*0*     *1*     *2*  nhidden[1]->npool[1]     */

  int batch_size = net->ninput.nstrides;
  int i, j, k, l;
  float *temp;

  if (cnn_mode && !(CNN_DEPTH==1 && FC_DEPTH==1) && !(CNN_DEPTH==3 && FC_DEPTH==1)
               && !(CNN_DEPTH==4 && FC_DEPTH==1) && !(CNN_DEPTH==6 && FC_DEPTH==2)) { /* IGNORE MACRO_PIPE in cnn_mode */
    if (THREAD > 0) exit(0);
  }
  if (eye_mode) { /* IGNORE MACRO_PIPE in eye_mode */
    if (THREAD > 0) exit(0);
  }
  if (spike_mode) { /* IGNORE MACRO_PIPE in spike_mode */
    if (THREAD > 0) exit(0);
  }
  if (spike_mode) {
    /* SMAX1 assumes -V* -C1 */
    // add bias broadcast<2>(hbias, hidden.shape);
    temp = net->nhidden[0].data;
    for (i=0;i<net->nhidden[0].nstrides;i++) {
      for (j=0;j<net->nhidden[0].nchannel;j++) {
        for (k=0;k<net->nhidden[0].kstrides*net->nhidden[0].stride_size;k++,temp++)
          *temp += net->hbias[0].data[j];
      }
    }
    relu4(&(net->nhidden[0]));                                              /* V1->c.nhidden[0].data */
    max_pooling(&(net->npool[0]), &(net->nhidden[0]), c[0].psize, c[0].psize); /*   ->c.npool[0].data */
    flat4Dto2D(&(net->nflat[0]), &(net->npool[0]));                         /*   ->f.nflat[0].data */
    smax_trial(0, net, c, f); /* IGNORE MACRO_PIPE */
  }
```

```
  else {
    int LANE = 0;
    for (l=0; l<CNN_DEPTH; l++) {
      /*****************************************************************************/
      if (!MACROPIPE || NTHREAD==1 || eye_mode) {}
      else if (THREAD == l*2) {
        if (THREAD>0) while (th_inference_args[THREAD-1].enq==th_inference_args[THREAD-1].deq) { inference_sigwait; }
                      while (th_inference_args[THREAD  ].enq!=th_inference_args[THREAD  ].deq) { inference_sigwait; }
        //th_inference_args[THREAD-1].deq = 1-th_inference_args[THREAD-1].deq; /* DBUF の場合 */
#if defined(EMAX7)
        if (l >= NLANE) {
          printf("nn_forward_CNN: LANE(%d) >= NLANE(%d)\n", l, NLANE);
          exit(1);
        }
        LANE = l;
        emax7[LANE].sigwait = 1; /* ON */
        emax7[LANE].sigstat = &th_inference_args[THREAD].stat;
        emax7[LANE].sigset  = &th_inference_args[THREAD].sigset;
#endif
      }
      else
        goto end_of_cnn; /* skip other task */
      /*****************************************************************************/
      if (l>0 || cnn_mode) {
        // first layer, conv, use stride=2
        /**********************ninput*** V CNN    */
#ifndef CUDA
/*★ IMAX3 ★*/conv_forward(THREAD, LANE, l==0?&(net->ninput):&(net->npool[l-1]), &(net->Ki2h[l]),
                          &(net->nhidden[l]), c[l].ksize, &(net->tmp_col[l]), &(net->tmp_dst[l]));
#else
        conv_forward_cuda(l==0?&(net->ninput):&(net->npool[l-1]), &(net->Ki2h[l]),
                          &(net->nhidden[l]), c[l].ksize, &(net->tmp_col[l]), &(net->tmp_dst[l]));
#endif
      }

      // add bias broadcast<2>(hbias, hidden.shape);
      temp = net->nhidden[l].data;
      for (i=0;i<net->nhidden[l].nstrides;i++) {
        for (j=0;j<net->nhidden[l].nchannel;j++) {
          for (k=0;k<net->nhidden[l].kstrides*net->nhidden[l].stride_size;k++,temp++)
            *temp += net->hbias[l].data[j];
        }
      }
      // Activation, relu, backup activation in nhidden
      // nhidden = F<relu>(nhidden);
      relu4(&(net->nhidden[l]));
      copy4D(&(net->nhiddenbak[l]), &(net->nhidden[l]));

      /*****************************************************************************/
      if (!MACROPIPE || NTHREAD==1 || eye_mode) {
      }
      else if (THREAD == l*2) {
        if (THREAD>0) th_inference_args[THREAD-1].deq = 1-th_inference_args[THREAD-1].deq;
                      th_inference_args[THREAD  ].enq = 1-th_inference_args[THREAD  ].enq;
      }
      /*****************************************************************************/
    end_of_cnn:
```

```
      /*****************************************************************************/
      if (!MACROPIPE || NTHREAD==1 || eye_mode) {
      }
      else if (THREAD == l*2+1) {
        while (th_inference_args[THREAD-1].enq==th_inference_args[THREAD-1].deq) { inference_sigwait; }
        while (th_inference_args[THREAD  ].enq!=th_inference_args[THREAD  ].deq) { inference_sigwait; }
        //th_inference_args[THREAD-1].deq = 1-th_inference_args[THREAD-1].deq; /* DBUF の場合 */
      }
      else
        goto end_of_maxpool; /* skip other task */
      /*****************************************************************************/

      // max pooling /*後段 nhidden が空いたら開始*/
      max_pooling(&(net->npool[l]), &(net->nhidden[l]), c[l].psize, c[l].psize);
      copy4D(&(net->npoolbak[l]), &(net->npool[l]));

      /*****************************************************************************/
      if (!MACROPIPE || NTHREAD==1 || eye_mode) {
      }
      else if (THREAD == l*2+1) {
        th_inference_args[THREAD-1].deq = 1-th_inference_args[THREAD-1].deq; /* NHIDDEN/NPOOL を使った DBUF の場合 */
        th_inference_args[THREAD  ].enq = 1-th_inference_args[THREAD  ].enq;
      }
      /*****************************************************************************/
    end_of_maxpool:
      continue;
    }
```

```
    /************************************************************************/
    if (!MACROPIPE || NTHREAD==1 || eye_mode)
      LANE = 0;
    else if (THREAD == CNN_DEPTH*2) {
      while (th_inference_args[THREAD-1].enq==th_inference_args[THREAD-1].deq) { inference_sigwait; }
      //th_inference_args[THREAD-1].deq = 1-th_inference_args[THREAD-1].deq; /* DBUF の場合 */
#if defined(EMAX7)
      if (CNN_DEPTH >= NLANE) {
        printf("nn_forward_FC: CNN_DEPTH(%d) >= NLANE(%d)\n", CNN_DEPTH, NLANE);
        exit(1);
      }
      LANE = CNN_DEPTH;
      emax7[LANE].sigwait = 1; /* ON */
      emax7[LANE].sigstat = &th_inference_args[THREAD].stat;
      emax7[LANE].sigset  = &th_inference_args[THREAD].sigset;
#endif
    }
    else
      goto end_of_fc; /* skip other task */
    /************************************************************************/
    for (l=0; l<FC_DEPTH; l++) {
      if (l==0)                                                 /***********************npool**** A CNN    */
        flat4Dto2D(&(net->nflat[0]), &(net->npool[CNN_DEPTH-1]));/************************|****** BOUNDARY */
      else                                                      /***********************nflat**** V FC     */
        copy2D(&(net->nflat[l]), &(net->nout[l-1]));

      // second layer full-connection
/*★ IMAX3 ★*/multiply_float2D(THREAD, LANE, &(net->nout[l]), &(net->nflat[l]), 0, &(net->Wh2o[l]), 0);
      repmat_add(&(net->nout[l]), &(net->obias[l]), batch_size);

      if (l < FC_DEPTH-1) {
        // activation, sigmloid, backup activation in fhidden
#if 1
        sigmoid(&(net->nout[l]));
#else
        relu2(&(net->nout[l]));
#endif
        copy2D(&(net->noutbak[l]), &(net->nout[l]));
      }
      else { /* l == FC_DEPTH-1 */
        // softmax calculation
        softmax2D(&(net->nout[FC_DEPTH-1]), &(net->nout[FC_DEPTH-1]));
      }
    }
    /************************************************************************/
    if (!MACROPIPE || NTHREAD==1 || eye_mode)   {
    }
    else if (THREAD == CNN_DEPTH*2)
      th_inference_args[THREAD-1].deq = 1-th_inference_args[THREAD-1].deq; /* NHIDDEN/NPOOL を使った DBUF の場合 */
    /************************************************************************/
    end_of_fc:;
  }

  if ((!MACROPIPE || NTHREAD==1 || eye_mode)
    ||(CNN_DEPTH==1 && THREAD==2)
    ||(CNN_DEPTH==3 && THREAD==6)
    ||(CNN_DEPTH==4 && THREAD==8)
    ||(CNN_DEPTH==6 && THREAD==12)
      ) {
    // copy result out
    copy2D( oubatch, &(net->nout[FC_DEPTH-1]));
  }
}
```

3.2 Chat GPT (vsim)

```
#undef  NCHIP
#define NCHIP 1
  check_lmm_fit++;
  int tmp;
  monitor_time_start(THREAD, IMAX_CPYIN);
  xmax_cpyin(3, i_m0A[LANE], &tmp, src0->data,    1, 1, 1, NRNB01d4,     1);
  xmax_cpyin(3, i_m0B[LANE], &tmp, params->wdata, 1, 1, 1, NBNB00xNE11d4,1);
  xmax_bzero(   i_m0C[LANE], NE01NE11);
  monitor_time_end(THREAD, IMAX_CPYIN);

  for (int ir = 0; ir < nr; ir++) { /* 5120, 20480, 50288 ■ */
    const uint8_t * restrict sd   = (const uint8_t *)((char *)i_m0A[LANE]+(ir*nb01));
    const uint8_t * restrict wd   = (const uint8_t *)((char *)i_m0B[LANE]);
    const uint8_t * restrict sdp[4];
    const uint8_t * restrict wdp[4];
    sdp[0] = sd   + sizeof(float)*1;
    wdp[0] = wd   + sizeof(float)*1;
    sdp[1] = sd   + sizeof(float)*2;
    wdp[1] = wd   + sizeof(float)*2;
    sdp[2] = sd   + sizeof(float)*3;
    wdp[2] = wd   + sizeof(float)*3;
    sdp[3] = sd   + sizeof(float)*4;
    wdp[3] = wd   + sizeof(float)*4;
    float         *       dst_col = (float *)        ((char *)i_m0C[LANE]+(ir*nb0));  /* nb0: 4B */

#define mul_mat_cores(r, c, d0, d1, d2, d3) \
        mop(OP_LDWR,  1, &BR[r][0][1], sd,          cofs, MSK_W1,  (Ull)sd, NBNB00d4,  0, 0, (Ull)NULL, NBNB00d4);\
        mop(OP_LDWR,  1, &BR[r][2][1], sdp[c],      cofs, MSK_W1,  (Ull)sd, NBNB00d4,  0, 0, (Ull)NULL, NBNB00d4);\
        exe(OP_FML3,     &d0,          BR[r][0][1],  EXP_H1010, BR[r][2][1], EXP_H1010, 0x0003000200010000LL, EXP_B5410, OP_NOP, 0LL, OP_NOP, 0LL);\
        exe(OP_FML3,     &d1,          BR[r][0][1],  EXP_H1010, BR[r][2][1], EXP_H1010, 0x0003000200010000LL, EXP_B7632, OP_NOP, 0LL, OP_NOP, 0LL);\
        exe(OP_FML3,     &d2,          BR[r][0][1],  EXP_H1010, BR[r][2][1], EXP_H1010, 0x0007000600050004LL, EXP_B5410, OP_NOP, 0LL, OP_NOP, 0LL);\
        exe(OP_FML3,     &d3,          BR[r][0][1],  EXP_H1010, BR[r][2][1], EXP_H1010, 0x0007000600050004LL, EXP_B7632, OP_NOP, 0LL, OP_NOP, 0LL)

#define mul_mat_corew(r, c, d0, d1, d2, d3, Force) \
        mop(OP_LDWR,  1, &BR[r][0][1], wd,          iofs, MSK_W1,  (Ull)wd, NBNB00xNE11d4, 0, Force, (Ull)NULL, NBNB00xNE11d4);\
        mop(OP_LDWR,  1, &BR[r][2][1], wdp[c],      iofs, MSK_W1,  (Ull)wd, NBNB00xNE11d4, 0, Force, (Ull)NULL, NBNB00xNE11d4);\
        exe(OP_FML3,     &d0,          BR[r][0][1],  EXP_H1010, BR[r][2][1], EXP_H1010, 0x0003000200010000LL, EXP_B5410, OP_NOP, 0LL, OP_NOP, 0LL);\
        exe(OP_FML3,     &d1,          BR[r][0][1],  EXP_H1010, BR[r][2][1], EXP_H1010, 0x0003000200010000LL, EXP_B7632, OP_NOP, 0LL, OP_NOP, 0LL);\
        exe(OP_FML3,     &d2,          BR[r][0][1],  EXP_H1010, BR[r][2][1], EXP_H1010, 0x0007000600050004LL, EXP_B5410, OP_NOP, 0LL, OP_NOP, 0LL);\
        exe(OP_FML3,     &d3,          BR[r][0][1],  EXP_H1010, BR[r][2][1], EXP_H1010, 0x0007000600050004LL, EXP_B7632, OP_NOP, 0LL, OP_NOP, 0LL)

  //EMAX5A begin mul_mat_q4_0_f32 mapdist=0
  /**/for (CHIP=0; CHIP<NCHIP; CHIP++) { /* will be parallelized by multi-chip (M/#chip) */
   /*2*/for (INIT1=1,LOOP1=ne11,rofs=MNBNB00_MNE0; LOOP1--; INIT1=0) { /* stage#0 *//* mapped to FOR() on BR[63][1][0] */
     /*1*/for (INIT0=1,LOOP0=nb,cofs=MBS; LOOP0--; INIT0=0) { /* stage#0 *//* mapped to FOR() on BR[63][0][0] */
           exe(OP_ADD,      &cofs,  INIT0?cofs:cofs,     EXP_H3210, BS,      EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL,                  OP_NOP, 0LL); /* stage#0 */
           exe(OP_ADD,      &rofs,  rofs,   EXP_H3210,   INIT0?NBNB00_NE0:0, EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL,                  OP_NOP, 0LL); /* stage#0 */
           exe(OP_ADD,      &iofs,  rofs,   EXP_H3210,   cofs,               EXP_H3210, 0LL, EXP_H3210, OP_AND, 0xffffffff00000000LL, OP_NOP, 0LL); /* stage#1 */
           exe(OP_ADD,      &oofs,  rofs,   EXP_H3210,   cofs,               EXP_H3210, 0LL, EXP_H3210, OP_AND, 0x00000000ffffffffLL, OP_NOP, 0LL); /* stage#1 */

           /* 1/4                                                                                                                                  */
           /* LDWR(sf:f32)   LDWR(s0:i4x8)                                    W W                  cofs iofs          4  #2  */
           /* FMUL sf(f32)*s0(f32) x w8                                       ~~~ 16  17  18  19    cofs iofs          6  #3  */
           /*                                                 16  17  18  19                                                                     */
           /* LDWR(wf:f32)   LDWR(w0:i4x8)                    |   |   |   | W W                  cofs iofs          8  #4  */
           /* FMUL wf(f32)*s0(w32) x w8                       |   |   |   | ~~~ 20  21  22  23    cofs iofs         10  #5  */
           /* FMUL sx(f32)*wx(w32) x w8         24  25  26  27                                      cofs iofs          6  #6  */
           /*                                   |   |   |   |                                                                                  */
           /* 2/4                               |   |   |   |                                                                                  */
           /* LDWR(sf:f32)   LDWR(s1:i4x8)      |   |   |   |               W W                    cofs iofs          8  #7  */
           /* FMUL sf(f32)*s1(f32) x w8         |   |   |   |               ~~~ 16  17  18  19     cofs iofs         10  #8  */
           /*                                   |   |   |   |   16  17  18  19                                                                  */
           /* LDWR(wf:f32)   LDWR(w1:i4x8)      |   |   |   |   |   |   |   | W W                  cofs iofs         12  #9  */
           /* FMUL wf(f32)*s1(w32) x w8         |   |   |   |   |   |   |   | ~~~ 20  21  22  23    cofs iofs         14  #10 */
           /* FMUL sx(f32)*wx(w32) x w8         28  29  30  31                                      cofs iofs          6  #11 */

           mul_mat_cores(2,  0, r16, r17, r18, r19);        /* stage #2-#3  */
           mul_mat_corew(4,  0, r20, r21, r22, r23, Force); /* stage #4-#5 */
           exe(OP_FML,      &r24, r16,          EXP_H3210, r20, EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#6 */
           exe(OP_FML,      &r25, r17,          EXP_H3210, r21, EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#6 */
           exe(OP_FML,      &r26, r18,          EXP_H3210, r22, EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#6 */
           exe(OP_FML,      &r27, r19,          EXP_H3210, r23, EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#6 */

           mul_mat_cores(7,  1, r16, r17, r18, r19);        /* stage #7-#8 */
           mul_mat_corew(9,  1, r20, r21, r22, r23, Force); /* stage #9-#10 */
           exe(OP_FMA,      &r28, r24,          EXP_H3210, r16, EXP_H3210, r20, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#11 */
           exe(OP_FMA,      &r29, r25,          EXP_H3210, r17, EXP_H3210, r21, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#11 */
           exe(OP_FMA,      &r30, r26,          EXP_H3210, r18, EXP_H3210, r22, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#11 */
           exe(OP_FMA,      &r31, r27,          EXP_H3210, r19, EXP_H3210, r23, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#11 */

           mul_mat_cores(12, 2, r16, r17, r18, r19);        /* stage #12-#13 */
           mul_mat_corew(14, 2, r20, r21, r22, r23, Force); /* stage #14-#15 */
           exe(OP_FMA,      &r24, r28,          EXP_H3210, r16, EXP_H3210, r20, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#16 */
           exe(OP_FMA,      &r25, r29,          EXP_H3210, r17, EXP_H3210, r21, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#16 */
           exe(OP_FMA,      &r26, r30,          EXP_H3210, r18, EXP_H3210, r22, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#16 */
           exe(OP_FMA,      &r27, r31,          EXP_H3210, r19, EXP_H3210, r23, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#16 */

           mul_mat_cores(17, 3, r16, r17, r18, r19);        /* stage #17-#18 */
           mul_mat_corew(19, 3, r20, r21, r22, r23, Force); /* stage #19-#20 */
           exe(OP_FMA,      &r28, r24,          EXP_H3210, r16, EXP_H3210, r20, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#21 */
           exe(OP_FMA,      &r29, r25,          EXP_H3210, r17, EXP_H3210, r21, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#21 */
           exe(OP_FMA,      &r30, r26,          EXP_H3210, r18, EXP_H3210, r22, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#21 */
           exe(OP_FMA,      &r31, r27,          EXP_H3210, r19, EXP_H3210, r23, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#21 */

           /* FAD tree */
           exe(OP_FAD,      &r3,  r28,          EXP_H3210, r29, EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#22 */
           exe(OP_FAD,      &r4,  r30,          EXP_H3210, r31, EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#22 */
           exe(OP_FAD,      &r2,  r3,           EXP_H3210, r4,  EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#23 */
           exe(OP_FAD,      &r1,  r2,           EXP_H3232, r2,  EXP_H1010, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#24 */
           exe(OP_NOP,      &AR[25][0], 0LL,    EXP_H3210, 0LL, EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL); /* stage#25 (dummy to set target location) */
           mop(OP_LDWR,  1, &r0,  dst_col,      oofs,      MSK_W0, i_m0C[LANE], NE01NE11,  0, Force, (Ull)NULL, NE01NE11); /* stage#25 */
           exe(OP_FAD,      &r0,  INIT0?r0:r0, EXP_H3210, r1,  EXP_H3210, 0LL, EXP_H3210, OP_NOP, 0LL, OP_NOP, 0LL);
           mop(OP_STWR,  1, &r0,  oofs,         dst_col,   MSK_D0, i_m0C[LANE], NE01NE11,  0, Force, (Ull)NULL, NE01NE11);
         }
       }
     }
  //EMAX5A end
     if (Force) Force = 0; /* reset wdat load to LMM */
   }
  //EMAX5A drain_dirty_lmm
```

Chapter 4

Appendix

4.1 References

This chapter lists related specifications, standards, references, related source programs, and tool chains.

- EMAX5 Basic patent proj-arm64/doc/pat35.tgz
- IMAX Basic patent proj-arm64/doc/pat36.tgz
- ARMv8 Architecture specification ..... proj-arm64/doc/arm/DDI0487A_f_armv8_arm.pdf
- ARM Cortex-A53 MPCore Processor Technical Reference Manual
  ..... proj-arm64/doc/arm/ARM-CORTEX-A53_R0P4.pdf
- ZYNQ Ultrascale+ SoC Technical Reference Manual
  ..... proj-board/zcu102/doc/ug1085-zynq-ultrascale-trm.pdf
- AMBA AXI4 and ACE Protocol Specification ..... proj-arm64/doc/arm/AXI4_specification.pdf
- IMAX3 Handbook proj-arm64/doc/emax7/emax7e.pdf
- IMAX3 Preprocessor proj-arm64/src/conv-c2d/conv-c2d
- IMAX2 Handbook proj-arm64/doc/emax6/emax6e.pdf
- IMAX2 Preprocessor proj-arm64/src/conv-c2c/conv-c2c
- IMAX2 SImulator proj-arm64/src/csim/csim
- Example(3x3-2D cnn) ..... proj-arm64/sample/mm_cnn_if/cnn+rmm.c
- Example(3x3-3D cnn) ..... proj-arm64/sample/mm_cnn_if/cnn3d+rmm.c
- Example(MM) ..... proj-arm64/sample/mm_cnn_if/mm+rmm.c
- Example(Inverse matrix) ..... proj-arm64/sample/mm_cnn_if/inv+rmm.c
- Example(Lightfield rendering) ..... proj-arm64/sample/mm_cnn_if/gather+rmm.c
- Example(Lightfield depth map) ..... proj-arm64/sample/mm_cnn_if/gdepth+rmm.c
- Example(Image recognition+stochastic ALU) proj-arm64/sample/tsim/imax.c
- Example(Chat GPT) proj-arm64/sample/vsim/imax.c