TranSoC: An Optimized Transformer SoC Design Based on The *Gemmini* Architecture on *Chipyard* Framework

Wentao Yao*, Hao Bai† and Liyang Qian†

Electronic & Computing Engineering Department, Zhejiang University - University of Illinois at Urbana-Champaign, Haizhou East Road, Haining, 314400, Zhejiang, China.

*Corresponding author(s). E-mail(s): Wentaoy.19@intl.zju.edu.cn;
Contributing authors: Haob.19@intl.zju.edu.cn; Liyang.19@intl.zju.edu.cn;
†These authors contributed equally to this work.

Abstract

Nowadays, transformer is a very popular deep learning model in both fields of Natural Language Processing and Computer Vision. With very large number of parameters in the model, transformer usually takes a significant amount of time for training and inference. Recent researches have been done to solve similar problems based on the Berkeley's systolic array generator called *Gemmini* architecture, and made great progress using this method. Thus, we follow this pattern and build a transformer accelerator based on the *Gemmini* architecture, which is mostly a matrix multiplication accelerator based on the systolic array technique. We developed the transformer testing framework for the *Gemmini* project with results. Additionally, we also provide our analysis for the *Gemmini* hardware and possible ways to implement the transformer accelerator.

Keywords: Chipyard, Gemmini, Transformer, System on Chip, Accelerator, Co-processor

1 Introduction

With the fast development of Machine Learning and Deep Learning, the computational resources becomes much more precious these days [1]. Recently, Google carried out its striking deep learning model called Transformer in the paper *Attention Is All You Need* [2], and has received up to 30,000 citations. This model requires a great amount of computational resources and takes a huge amount of time for running experiments, so lots of improvements on tweaking the model has been made to reduce the time for training the model [3], and there are also lots of researches made to improve the efficiency between the software and existing hardwares, like *TensorFlow* [4] and *Pytorch* [5].

On the hardware side, there have also been lots of researches to accelerator the computing models like an agile systolic array generator enabling systematic evaluations of deep-learning architectures called *Gemmini* [6] and an open-source deep neural network (DNN) accelerator which has received a lot of attention by the community since its introduction by Nvidia called *NVDLA* [7]. Excellent frameworks to support such hardware design is also implemented like the integrated design, simulation, and implementation framework for custom SoCs called *Chipyard* [8].

However, despite the great success of reducing the computational resources the Transformer requires, and also the performance break-though that hardware design achieves, there has not been such a hardware orienting Transformers. Thus, we introduce our TranSoC design - an optimized Transformer SoC design based on the *Gemmini* architecture on the *Chipyard* framework.

2 Transformer Testing Framework

Our transformer model is based on Google's classical paper "Attention is All you need". The transformer model is a kind of "encoder-decoder" structure using "Attention" layer to take place of CNN or RNN. Now, this model is widely used in the NLP field. Also, in computer vision and other deep learning field, the transformer model also achieve very good results. The architecture of Transformer is shown by Fig. 1.

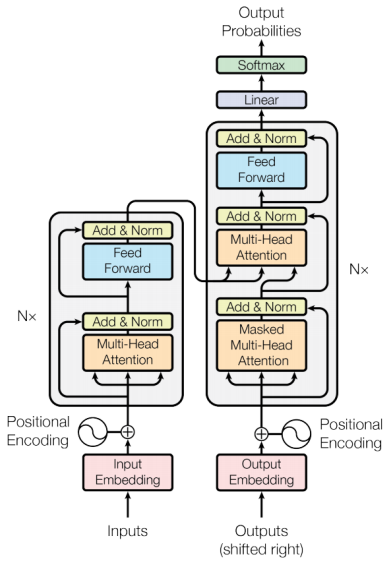

Fig. 1 The architecture of Transformer.

2.1 Positional Encoding

Since transformer model can process all the input simultaneously. We need some methods to explain the order of the word vector input. Here we use an additional input vector to the transformer input to solve this problem. We use the method from Google's paper and express in Formula 1 and 2.

$$PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}}) \quad (1)$$

$$PE(pos, 2i+1) = \cos(pos/10000^{2i/d_{model}}) \quad (2)$$

By using this method, any position PE_{pos+k} can be represented as the linear function of PE_{pos}. Besides, the triangular function will not be limited by the size of our dictionary. For this part, it actually contains the triangular function and a simple matrix add. We have not finished this part in the hardware module, so, we just use the Taylor expansion to realize the function of the triangular function.(This part is actually operated by the RISC-V CPU, has nothing to do with the accelerator).

2.2 Multi-head Attention

The Multi-head Attention is the most important part in the transformer. The attention mechanism let the module to "learn" the relationship between the words in the sentences.The attention mechanism in the transformer actually have two parts, self-attention and encoder-decoder attention. These two kinds of attention is used in transformer encoder and decoder separately. Since the only difference between the two attention is the input of them. So, we will use self-attention to explain this part.

Firstly, we need to get the matrix Q, K, V, which stands for the matrix of Query, Key and Value. The calculation process is shown in Formulas 3, 4 and 5.

$$Q = X \cdot W^q \quad (3)$$

$$K = X \cdot W^k \quad (4)$$

$$V = X \cdot W^v \quad (5)$$

In the formula, W^q, W^k, W^v represent the weights that muliplied with the input X. In our test code, we use three tiled matrix multiplication to realize this function. The code is shown in Appendix A.1.

After that, we will calculate the score of self-attention. This value represent that when considering a word in the specific position from the sentence, its attention degree to other parts of the sentence. This is calculated by the dot product of

Query and Key. Then, its value will be divided by a constant. Usually, this constant is the square root of the dimension of the input matrix. Then, After applying the $Softmax$ function, we get the result, which means the degree of correlation of each word in the sentences to the rest of the words. At the end, we multiply the Value V with the result after $Softmax$ function, we get the result of Self-attention. The principles are shown in Fig. 2.

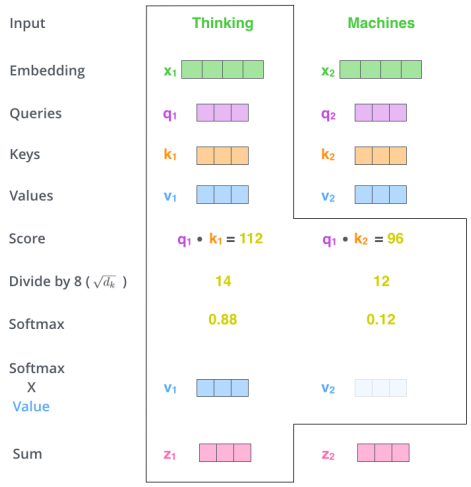

Fig. 2 The graph illustration for the process of self-attention.

Therefore, if we represent the output of attention as A_o, the attention process as $\mathcal{F}$, the softmax function as $\mathcal{S}$, then we can write the formula as a single expression:

$$A_o = \mathcal{F}(Q, K, V) = \mathcal{S}(\frac{QK^T}{\sqrt{d_k}})V \tag{6}$$

Now, for the multi-head attention, it is very similar to the self-attention. The difference is that multi-head attention will calculate several Q, K, V matrix rather than one for each. Then, it will generate several different attention outputs. We then concatenate these different output to one single matrix and send it to the feed forward layer, we will get final output. The graphical illustration is shown in Fig. 3.

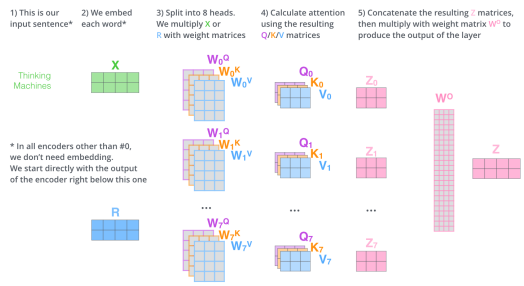

Fig. 3 The graph illustration for the process of Multi-head attention

2.3 Softmax

The Softmax function $\mathcal{S}$ is a very important function in transformer, and also many deep learning models. This function can map the output of the neural network layers to the range 0 to 1. Thus, if we represent x as the input vector, h as the weight (parameters) of the neural network, and p as the normalized output probability, then it has the form shown in Formula 7.

$$P(y \mid x) = \frac{e^{h(x,y_i)}}{\sum_{j=1}^{n} e^{h(x,y_i)}} \tag{7}$$

From the perspective of the hardware, it is clear that the exponential function takes the largest cost of computation. So, we need a better way to improve this calculation process. In our code, we use piece-wise linear function to fit the exponential function. Here, since we have not finish the hardware module of this part, we use the test code of transformer to simulate this process. The code is shown in Appendix A.2.
Our method is that we divide the region of the derivative of the exponential function to n parts, so that we can write n piece-wise linear function to fit the exponential function . Let's assume the domain of the exponential function to be $x \in [a, b]$. Since the derivative of the exponential function is equal to itself. So, we can write its derivative in the domain $f'(x) \in [e^a, e^b]$. If we divide it into n parts, each piece-wise linear function have the derivative as shown in Formula 8.

$$sf'(a + i \cdot \Delta x) = e^{a+i \cdot \Delta x} \tag{8}$$

$$\Delta x = \frac{b-a}{n} \tag{9}$$

$$i = \{0, ..., n-1\} \tag{10}$$

Therefore, we can write the piece-wise linear function as shown in Formula 12 and exponential function in Formula 13.

$$a_+ = a + \Delta x \cdot i \tag{11}$$
$$y_l = f'(a_+) \cdot x + f(a_+) - f'(a_+) \cdot (a_+) \tag{12}$$
$$y_e = e^{a+\Delta x \cdot i} x + (a + \Delta x \cdot i - 1) e^{a+\Delta x \cdot i} \tag{13}$$

2.4 Addition and Normalization

This is the residual module originated in the ResNet. It is a very useful layer in the machine learning module. Here, we use the basic matrix multiplication and matrix addition function this part. For the normalization, we use Layer Normalization rather than Batch normalization. Since Layer Normalization performs better than Batch Normalization in NLP tasks. Here, we have the formula of the layer normalization as shown in Formula 14.

$$LN(x_i) = \alpha \cdot \frac{x_i - \mu}{\sqrt{\sigma_L^2 + \epsilon}} + \beta \tag{14}$$

The most costly part of this calculation is the square root. Here, we use the same method as the exponential function, that we transfer the square root to the piece-wise linear functions. As the method is the same, so we omit the process of derivation. The corresponding code is shown in Appendix A.3.

2.5 Masked Multi-head attention

The masked multi-head attention is in the decoder of transformer. It has two main functions. Firstly, it can eliminate the influence from any kinds of padding. Secondly, it will cover some part from input, so that the decoder will not see the predicted part. As the word from position t will not see the information from the word in position $t+1$ in seq-to-seq model. We use the lower triangle matrix to realize that. All the value in the upper triangle will be $-\infty$, so that after $Softmax$ function, its value will be 0. To realize that, we use code to generate the lower triangle matrix and then use the basic matrix multiplication to make that.

3 Experimental Results

If you want to reproduce our code, please refer to our GitHub Repository at https://github.com/BiEchi/chipyard. The repository consists of the code architecture we construct for testing the efficiency of the SoC running Transformer, and experimental results, and also our explanation of the source code of the architecture of Gemmini (which will also be mentioned in Section 4).

For our experimental result running Transformer, we used the *Verilator* to simulate our transformer test codes. The configuration is that, we have word dimension 512, sentence length 128. Thus, if we represent the matrix dimension (size) A as MD-A, matrix dimension B as MD-B. cycle in WS dataflow as #C-WS, cycle in OS dataflow as #C-OS, and the output matrix dimension as MD-O, then the result of the *Gemmini* accelerator in both OS and WS dataflow are shown in Table 1.

Table 1 Experimental results of basic matrix multiplication.

MD-A	MD-B	#C-WS	#C-OS	MD-O
16*16	16*16	10	15	16*16
1*16	16*16	10	15	1*16
16*16	16*1	12	15	16*1
16*24	24*16	23	26	16*16

Source: This experimental result was carried out by *Verilator*, and can be reproduced by our code on GitHub.

Following the same pattern, we run our model on the encoder and decoder in Transformer separately, and we gain the number of cycles our model runs to carry it out as shown in Table 2.

Table 2 Experimental results of transformer.

	C-OS Dataflow	C-WS Dataflow
Encoder	320522	289540
Decoder	5959402	5882893

Source: This experimental result was carried out by *Verilator*, and can be reproduced by our code on GitHub.

4 Gemmini Framework Analysis

The *Gemmini* matrix multiplication accelerator is described by the *HDL* Chisel, which is very similar to the *HDL* Verilog. The content below records what we have analyzed about the code of *Gemmini* source code.

4.1 Controller

The responsibility of Controller Moudle is to unfold the ROCC instruction, which will be loaded into a buffer. Then, the ROCC command will be unfolded into a series of load, store and execute instructions. Finally, it will enter into the Re-Order Buffer(ROB) unit.

The ROB unit sends the reordered instruction to *load-controller* unit, *store-controller* unit and *execute-controller* unit. The load-controller and store-controller respectively interact with the ***dma*** engine of *Spad* unit, a on-chip memory of Gemmini. The *dma* engine interacts with the *reader unit* and *writer unit* according to the read and write instruction from corresponding controller unit. The *reader unit* and the *writer unit* finish the assignment of transferring data from *DRAM* to *Spad banks* by *TLB* (translation look aside buffer), which is a look-up table to store the mapping between virtual address and physical address. The graphical illustration of CU is shown in Fig. 4.

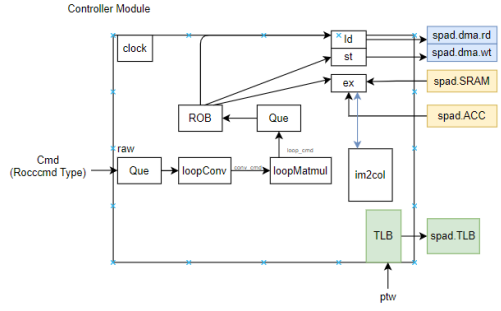

Fig. 4 The graphical illustration of the Control Unit design.

4.2 Load Controller

Load Controller is very similar to the construction of Store Controller. Their running mode is very similar.Load Controller unit uses FSM to control its running mode, which is *waiting for command waiting for dma request* and *sending row*. The responsibility of Load Controller unit is to load the data from Spad into systolic-array unit or anywhere else.

In *waiting for cmd* state, the load controller chooses to config itself or implement loading assignment according to the *func* arguments in ROCC command. Then if the choice is to implement loading instruction, the load controller will firstly load the instruction into *cmd tracker*, which will record the total data required to be loaded into *spad* unit since sometime the data is too large to load it entirely into *spad* unit at one time. If the *cmd tracker* is full, the controller will be suspended until space appears to store the instruction. After storing the instruction into *cmd tracker*, the tracker will return a specific id of this instruction to label it. Finally, the FSM enters into the *waiting for dma request* or *sending row* states according to whether the *Spad dma* engine has finished dealing with the read operation or not.

In *waiting for dma request* state, the controller unit will be suspended until *Spad dma* engine has finished reading operation.

In *sending row* state, the controller will keep running until the last row has been read from *Spad unit*. There is a counter inside *Load controller* to confirm whether the row, which is read, is the last one. The design of the load controller is shown in Fig. 5.

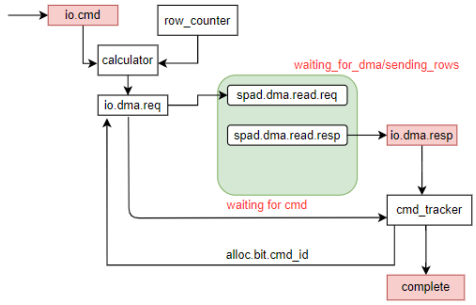

Fig. 5 The graphical illustration of the Load Controller design.

4.3 Spad Unit

The responbility of *Spad* unit is to store all data generated during the process of calculation, including data like the source data of matrix A and matrix B, the intermediate accumulate result and so on. The IO port of *Spad* unit consists of *dma*, *Sram*, *Acc* and *TLB* unit. *dma*, *Sram* and *acc* units are used to deal with respective requests from outside hardware, for example, the read request from *Load Controller*. These units contains request part and response part, the former used to deal with request, the latter used to return the value.

There are tow kinds of banks working as memory to store data,*Spad banks* and *acc banks*. Spad banks interact with *Sram unit* and acc banks interact with *acc unit*. Then both of them will send the result to outside hardware by **.resp* port. What's more, the output data will also be sent to *writer* unit to copy the data into DRAM. In conclusion, the Spad unit works more like a cache on the Gemmini. The Spad banks contain 4 banks, respectively serving for storing data in matrix *A,B,C* and *D*. The graphical illustration of the Spad IO and banks design is shown in Fig. 6.

dma unit interacts with the reader and writer to affect DRAM according TLB. When there is a requst for vector multiplication, the result from reader unit is sent to *vector multiplication* unit to do vector multiplication calculation. Then the intermediate result will be store into *acc bank* or *spad bank*. And the data in a bank will be flushed if there is a read request from dma engine. The graphical illustration of the Spad DMA workflow is exhibited by Fig. 7.

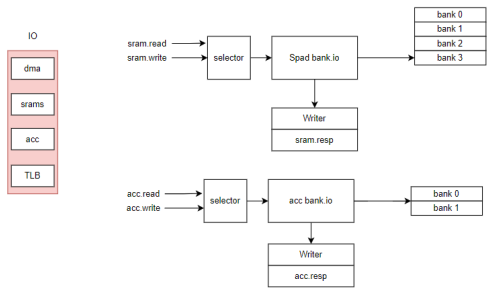

Fig. 6 The graphical illustration of the the Spad IO and banks design.

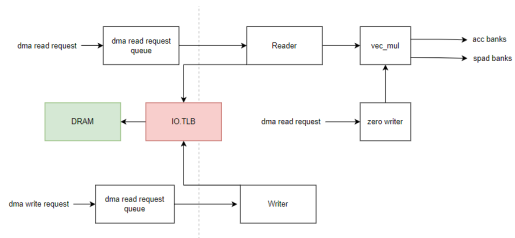

Fig. 7 The graphical illustration of the Spad DMA workflow.

4.4 Execute controller

The responsibility of Execute Controller is to execute matrix multiplication. The execute controller also uses FSM to implement. The FSM contains *waiting for cmd* and *compute* states. In *waiting for cmd* state, the unit decides to configure itself or do the assigned computation according to the argument of command. In *computation* state, the unit does one of the *overlap computation*, *single multiplication computation* and *Do pre-loads* until the systolic-array's calculation has been finished.

The *Multi-head-cmdQue* unit is a cmd buffer. The input cmd is unfolded after entering into the *TransposePreloadUnroller* unit, then it will be stored into the Multi-head-cmdQue buffer, The data info of command will be transferred into the *Req* port of IO to extract data from *Spad* unit. The other data in a unrolled command will be interpreted into corresponding control signal to control the performance of systolic-array calculation unit. And the data from Spad unit's unit will be loaded into the systolic-array unit. The graphically illustration of the Execution Controller is shown in Fig. 8.

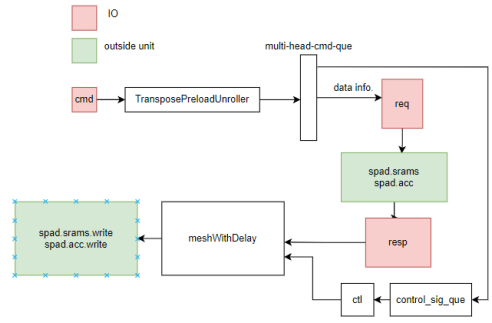

Fig. 8 The graphical illustration of the Execute Controller design.

5 Theoretical Amelioration on Implementation

In this part, we talk about the theoretical amelioration on how to make the SoC more suitable for Transformer calculations. The implementation is not carried out yet, and will probably be a project for Zongsheng, Wang[1] as a deeper insight of design.

5.1 Design Challenges and Solutions

If we want to ameliorate our design, the chip we design need to:

- Improve the ML performance than *Gap 9*.
- Decrease the area (after scale) of the chip than *Gap 9*.
- The HWCE (A cluster of 8 cores with an architecture optimized for the execution of vectorized and parallelized algorithms combined with a specialized Convolutional Neutral Network accelerator) should obtain a wider bandwidth.

The basic ideology to solve these challenges is listed below:

- Get rid of MRAM and FPU, and increase L2 memory.
- Set base on area-saving systolic array.
- Use a look-up table to calculate the activation function.

In details, the difference between our design and *Gap 9* is shown in Fig. B1 in Appendix B.

5.2 Analysis

5.2.1 Analysis on *Gap 9*

According to the official description of *Gap 9*[2], if we denote g as number of grants per master and c as number of cycles where a master asserts the request, we gain the Formula 15.

$$P_{gnt} = \frac{g}{c}. \tag{15}$$

5.2.2 Analysis on the Systolic Array

According to the ratio of *Vega* the size of a 16*16 systolic array is approaching half the size of the cluster domain. Thus, to simplify 8 parallel RISC-V, optimize L1 access structure and get rid of APU-FPU are necessary.

As we see in Fig. 9, most of the area on a SoC is spent on memory. Moreover, because the number of arguments decides the area, we deduct that the number of arguments in the learning process decides the area of the SoC.

Design	Freq (MHz)	Floorplan	Area (mm^2)	Power (mW)	Setup WNS (ps)	Setup TNS (ps)
16×16	500	Block	1.34	321.71	1	0
16×16	500	Semi-Ring	1.21	312.41	0	0
16×16	1000	Block	1.34	773.70	-12	-235
16×16	1000	Semi-Ring	1.21	766.12	-14	-420
32×32	500	Block	2.81	1058.24	-14	-100
32×32	500	Semi-Ring	3.01	1078.71	-9	-59
32×32	1000	Block	2.81	2796.51	-530	-2716
32×32	1000	Semi-Ring	3.01	2683.01	-315	-508

Fig. 9 The comparison of *Gap 9* and our chip.

5.2.3 Conclusion on Area

After we remove the 2MB MRAM, the die are becomes approximately 3mm*3mm, which is much smaller than the size of *Gap 9*, which is 3.7mm*3.7mm. This proves that *Gap 9* increased the performance of HWCE.

Directly transplanting *Gap 9* to 28mm process will cause the area up to 4.7mm*4.7mm, which shows that the HWCE that *Gap 9* uses is more complicated than the systolic array.

Thus, we can optimize the PE (Processing Elements) to use more area to gain less power consumption, then save some area on other parts of the SoC to compensate for the exceeding 4mm*4mm part to fit in the 28nm node and achieve area restraint.

5.3 Solutions

We use *Vega* to show how we can truncate the SoC and build a new one. The original figure is shown in Fig. B2 and the new figure is shown in Fig. B3, both in Appendix B.

- Remove MRAM, merge the area saved for L2 memory to gain L2 memory up to approximately 3.6MB.
- Remove the floating point unit of FC.
- Remove all external interfaces except CPI & CSI.
- Remove FPU in the cluster, and make the 8 parallel RISC-V a whole SIMD device.

[1] Wang is also the first designer of the theoretical amelioration part too.

[2] http://asic.ethz.ch/2020/Vega.html

- HWCE is required to provide 2 acceleration functionalities: GEMM (including the convolution calculations and matrix multiplication operations operations after the img2col operation) and the element-wise non-linear operators (using LUT).
- Modify the standard structure of HWCE and gain internal private cache.

More specifically, about the architecture of HWCE, we show in Fig. 10. The overall pattern of the tiles is input activation broadcast, partial sum systolicly push-down and weight pre-loading, while we can fine-tune the architecture according to the requirements.

The private Spad with 64 KiB size is used to cache the Feature Map to strengthen the optimization of residual connection and the bandwidth of **concat**.

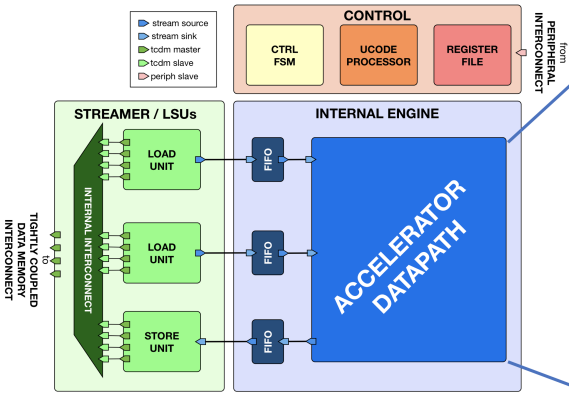

Fig. 10 The new architecture of HWCE.

One more amelioration is that we use GEMM to deal with the Depth-wise Convolution. Basically, we have 2 blue prints. The details will be provided in future work.

6 Conclusions and Future Work

Our research aims to optimize the hardware construction of Gemmini matrix multiplication accelerator so that we can decrease the time to calculation in vector-vector multiplication form or calculation in vector-matrix multiplication, and we can finish the ML assignment with less power consumption.

The design our leader commits theoretically decreases the area of SOC chips and improves the efficiency of convolution calculation. However, because all of our work is on computer and code, what we have done is just the simulation cases of final hardware. The Gemmini simulator we use do not contain the counter to record the time used for matrix multiplication. Therefore, future work needs to be done to build our virtual design and measure its performance in application.

The future work of our research includes implementing the hardware designs using the methods we mentioned in Section 5 and test it on the framework we've already developed in Section 2, and complete more code analysis about *Gemmini* as mentioned in Section 4.

7 Acknowledgements

We need to sincerely thank Professor Kejie Huang and his graduate student Zongsheng Wang. Zongsheng Wang is also the original designer of section 6. Without the help from Wang and Prof. Huang we can never achieve developing the framework and construct our workflow.

References

[1] E. J. Horvitz, "Reasoning about beliefs and actions under computational resource constraints," *arXiv preprint arXiv:1304.2759*, 2013.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[3] S. Anju, D. Mishra *et al.*, "Faster training of edge-attention aided 6d pose estimation model using transfer learning and small customized dataset," in *2021 Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. IEEE, 2021, pp. 62–67.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in

12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), 2016, pp. 265–283.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[6] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister *et al.*, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," *arXiv preprint arXiv:1911.09925*, 2019.

[7] F. Farshchi, Q. Huang, and H. Yun, "Integrating nvidia deep learning accelerator (nvdla) with risc-v soc on firesim," in *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE, 2019, pp. 21–25.

[8] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

Appendix A Code Space

A.1 The code for transformer multiplication and calculation of matrix Q,K,V.

```
for (int count = 0; count < n_head; count++)
  {
    tiled_matmul_auto(wordNum, weightDim, wordDim,
                      (elem_t *)word_vector,
                      (elem_t *)q_mats[count], NULL,
                      (elem_t *)z_qs[count],
                      wordDim, weightDim,
                      weightDim, weightDim,
                      MVIN_SCALE_IDENTITY,
                      MVIN_SCALE_IDENTITY,
                      MVIN_SCALE_IDENTITY,
                      NO_ACTIVATION, ACC_SCALE_IDENTITY,
                      0, false,
                      false, false,
                      false, false,
                      3,
                      accel_type);
    tiled_matmul_auto(wordNum, weightDim, wordDim,
                      (elem_t *)word_vector,
                      (elem_t *)k_mats[count], NULL,
                      (elem_t *)z_ks[count],
                      wordDim, weightDim,
                      weightDim, weightDim,
                      MVIN_SCALE_IDENTITY,
                      MVIN_SCALE_IDENTITY,
                      MVIN_SCALE_IDENTITY,
                      NO_ACTIVATION, ACC_SCALE_IDENTITY,
                      0, false,
                      false, false,
                      false, false,
                      3,
                      accel_type);
    tiled_matmul_auto(wordNum, weightDim, wordDim,
                      (elem_t *)word_vector,
                      (elem_t *)v_mats[count], NULL,
                      (elem_t *)z_vs[count],
                      wordDim, weightDim,
                      weightDim, weightDim,
                      MVIN_SCALE_IDENTITY,
                      MVIN_SCALE_IDENTITY,
                      MVIN_SCALE_IDENTITY,
                      NO_ACTIVATION,
                      ACC_SCALE_IDENTITY,
                      0, false,
                      false, false,
```

```
                        false, false,
                        3,
                        accel_type);
    }
```

A.2 The code for the exponential function mentioned in the Softmax calculation part.

```
float my_exp(elem_t number)
{
  // float exp;
  // a is the slope
  float slope[24] = {0.006737946999085467, 0.0717489450711988,
  0.13736897328580006, 0.20417102465724862, 0.27276942684711947,
  0.34384344502557984, 0.41816603105197925, 0.49664045641486215,
  0.580348760164344, 0.6706178960296387, 0.7691126983842523,
  0.8779702723670761, 1.0000000000000004, 1.1389907283579461,
  1.3001995703630875, 1.4911621147011624, 1.7231018116017323,
  2.0135290773908743, 2.39139462735484, 2.9083003165164474,
  3.666100015528776, 4.897854637692817, 7.279664221697798,
  13.937487150614775};
  // b is the inter
  float x0[24] = {-5, -3.38975777788707, -2.274768975664783,
  -1.773836174919944, -1.4369560632507996, -1.1788714089484582,
  -0.9665265685144582, -0.7834156082000947, -0.6199855288515437,
  -0.4701006376974108, -0.3294748672686874, -0.19487414204749406,
  -0.06366558341604925, 0.0664754970873836, 0.19778114058929605,
  0.3325900427376481, 0.4735679272763055, 0.6240101415956079,
  0.7883219072584459, 0.9728773115664778, 1.187761027381711,
  1.4508594080418584, 1.7998227092372945, 2.3442268793669405};
  float y0[24] = {0.3860500834856052, 0.017587673747372003,
  0.0975869440730206, 0.1663995685965194, 0.23518072618265692,
  0.30557832941474355, 0.37859171088297716, 0.45516249439975104,
  0.5363284835993326, 0.6233139943480013, 0.7176201526404586,
  0.821143279619808, 0.936340493578349, 1.066481574081782,
  1.216037484611534, 1.391315961265941, 1.6015368417007017,
  1.8607640937356675, 2.191610611655227, 2.632955413966755,
  3.257901792686428, 4.22244677011019, 5.931618293254853,
  9.894697852690719};
  for (int i = 0; i < 24; i++)
  {
    if (number < x0[i])
    {
      if (i != 0)
      {
        return slope[i - 1] * (number - x0[i - 1]) + y0[i - 1];
      }
      else
      {
```

```
        return slope[0] * (number - x0[0]) + y0[0];
      }
    }
  }
  return slope[23] * (number - x0[23]) + y0[23];
}
```

A.3 The code for the square root function mentioned in the layer normalization calculation part.

```
float my_sqrt(elem_t number)
{
  float sqrt;
  // a is the slope
  float a[16] = {0.3549, 0.1470, 0.1128, 0.0951, 0.0838,
  0.0758, 0.0697, 0.0648, 0.0609, 0.0576, 0.0548, 0.0523,
  0.0502, 0.0483, 0.0466, 0.0451};
  // b is the inter
  float b[16] = {0, 2.8174, 3.9843, 4.8798, 5.6347, 6.2998,
  6.9011, 7.4540, 7.9687, 8.4521, 8.9093, 9.3441, 9.7596,
  10.1581, 10.5416, 10.9116};
  float step = 7.9438;
  for (int i = 0; i < 16; i++)
  {
    if ((step * i <= number) && (number < step * (i + 1)))
    {
      sqrt = a[i] * number + b[i];
      break;
    }
  }
}
```

Appendix B Big Graph Space

	Gap9	Our Chip
Processing Power	150.8GOPS (32.2GMACs ML & 15.6GOPs DSP)	百GOPS
Power Efficiency	0.33mW/GOP	< 1 W
L1 Memory	128kB	128kB + banked (16 way +)
L2 Memory (RAM)	1.5MB （应为1.6MB）	3MB +
L3 Memory (NVM)	2MB	不需要
External Memory	2x QSPI/OCTO-SPI/HyperBus/SDIO	需要
FC Frequency	400MHz	FPU removed
Cluster Frequency	400MHz (8 RISC(parallel) + 1RISC(master) + 1 APU(Float) + 1 HWPE(Vega: 3 * conv3 engine(3*3 PEs)))	HWPE -> 16*16 PEs WS @ over 500MHz(Area) Or 16*16 PEs WS @ 200MHz(power) * N cluster APU -> LUT based activation 8RISC -> 8 wide SIMD ? 1RISC master(optional)
Fix Point Precision	8,16,32,64-bit	4，8，16
Floating Point Precision	16, 32	V0 不需要
Sound Interface	3 master/slave SAI full duplex, I2S and TDM 4/8/16 ch capable	不需要
Camera Interface	8-bit CPI, 2-lane CSI-2	需要
Package type	WL-CSP 3.7mmx3.7mm - BGA 5.5mm x 5.5mm 制程工艺scale后等效28nm面积（WL-CSP）：4.7mm*4.7mm	Area < 4*4
Process	22 nm	28 nm

Fig. B1 The comparison of *Gap 9* and our chip.

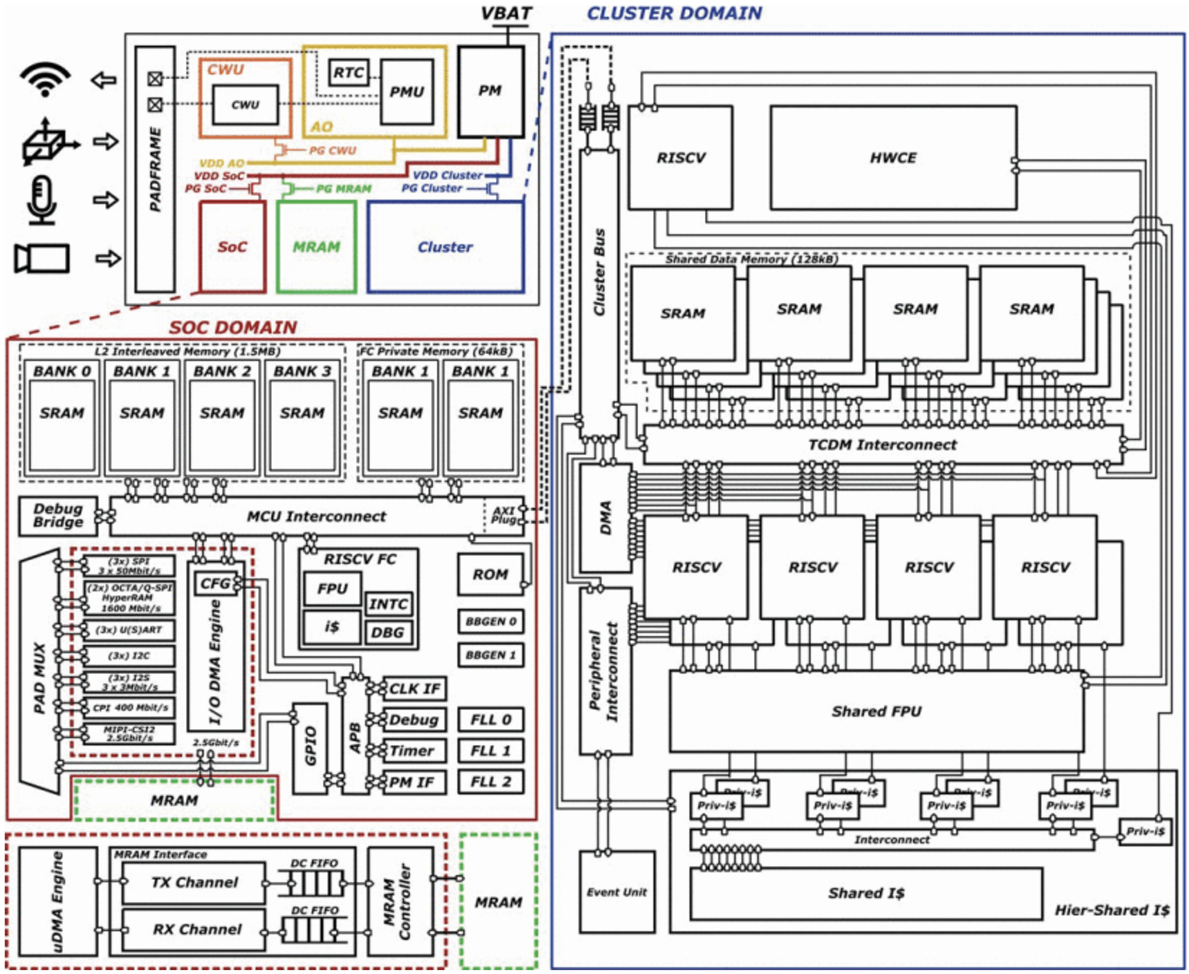

Fig. B2 The original SoC Layout.

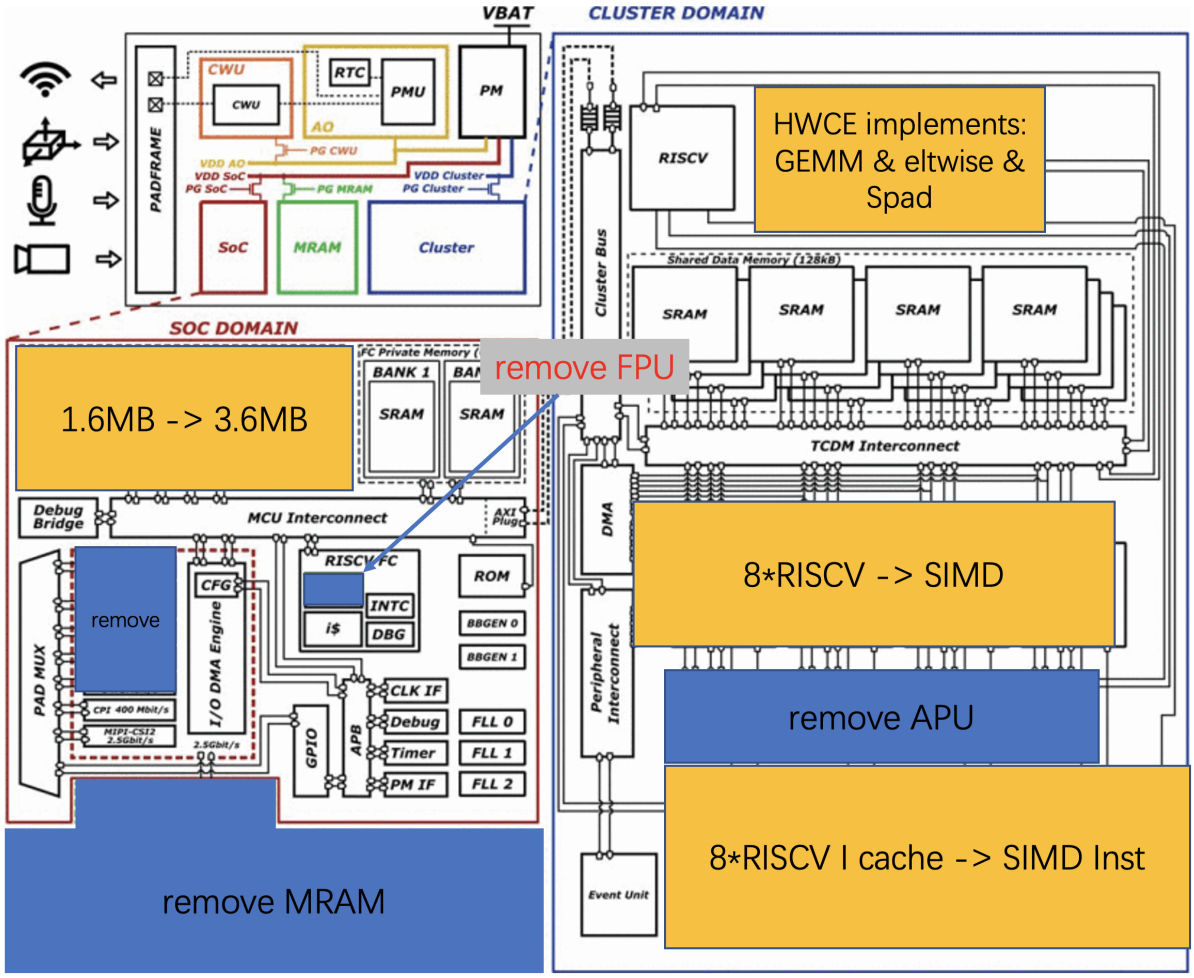

Fig. B3 Our design to refine the original SoC Layout.