Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Fall Semester 2022

Efficient Execution of Transformers in RISC-V Vector Machines with Custom HW Acceleration

Master's Thesis

Xiaorui Yin
yinx@student.ethz.ch

15.02.2023

Advisors: Matteo Perotti, ETH Zurich, mperotti@iis.ee.ethz.ch
Renzo Andri, Huawei Technologies, renzo.andri@huawei.com
Victor Jung, ETH Zurich, jungvi@iis.ee.ethz.ch

Professor: Prof. Dr. Luca Benini, lbenini@iis.ee.ethz.ch

Abstract

Transformer is a very popular deep learning model that has achieved great success in various tasks. Consequently, the hardware used for its acceleration has become a topic of significant interest. However, the computational demand of transformer is challenging for traditional CPUs, while specialized accelerators, although capable of delivering optimal performance and efficiency, are inherently limited in terms of flexibility. In order to preserve a high degree of flexibility, we use a general-purpose vector processor, Ara, to accelerate the transformer network. The objective of this project is to explore the capability and challenge of using a general-purpose vector processor, Ara, and optimize the performance from both hardware-software levels. We first vectorize the transformer and execute it on Ara. Through an analysis of performance results, we propose an improved execution of MatMul with new vector instructions and hardware that is more scalable and exhibits superior performance in small MatMul. Moreover, we also transpose all activation matrices to enhance other kernels. As a result, the transformer experiences a performance enhancement ranging from 5% to 17% in the case of chaining and from 17% to 125% in the case of decomposing, and a 10% to 11% increase in energy efficiency. Small MatMul operations have a 2.5x greater utilization with an improvement in energy efficiency reaching up to 76%.

Acknowledgments

I would like to thank all the individuals who have contributed to the successful completion of my master's thesis. My heartfelt thanks go to my advisors Matteo Perotti, Victor Jung, and Dr. Renzo Andri for their patient and meticulous guidance throughout the project. Not only did they provide me with technical advice, but they also imparted invaluable lessons in presentation and essay composition. I am also grateful to Prof. Dr. Luca Benini for providing me with the opportunity to complete my challenging project in his laboratory and for his insightful advice. I would also like to express my appreciation to Dr. Frank Kagan Gürkaynak for his technical support and administrative assistance. I am grateful to my friends and labmates in Zurich, with whom I have shared numerous cherished memories. Finally, I would like to express my gratitude to my parents and my younger sister, who have consistently showered me with love and encouragement. Over the years, they have been my steadfast support, and I shall never forget it in my life.

Declaration of Originality

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Efficient Execution of Transformers in RISC-V Vector Machines with Custom HW Acceleration

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Xiaorui	Yin

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 13. Feb. 2023	Xiaorui Yin

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

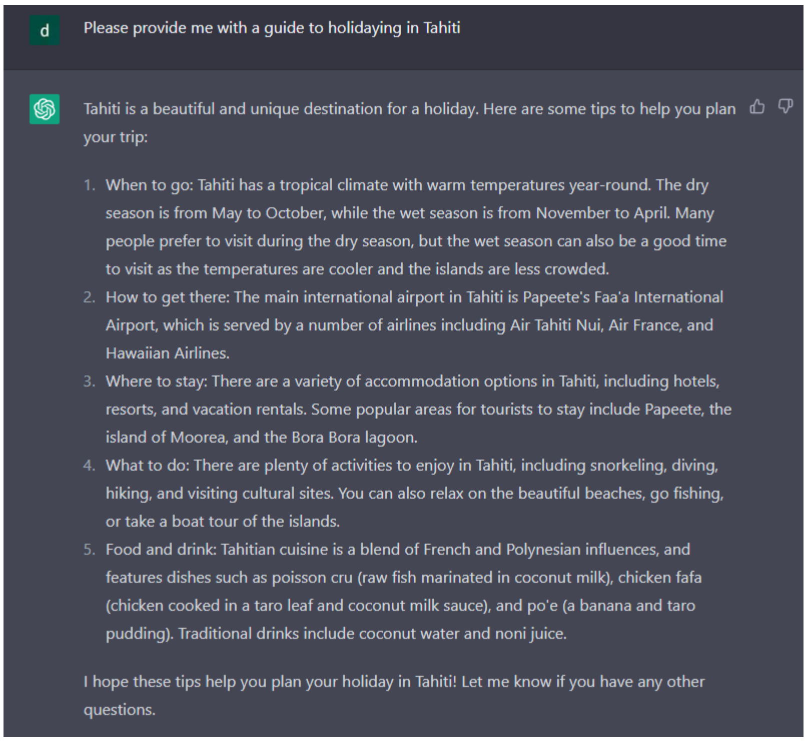
Contents

Chapter 1

Introduction

In 2022, a chatbot called ChatGPT [1] burst into the public eye and gained widespread attention. It became the first AI product to be widely acclaimed among mainstream users, and people from all walks of life are enthusiastically sharing their marvelous conversations with ChatGPT. It amazes people with its conversational capabilities, not only answering questions with the human logic and knowledge used for training, but also acting as a translator for different languages. In addition, it can also write code based on the user's description just like Copilot. The underlying language model behind ChatGPT is GPT-3 [2], the most powerful language model by far, which performs stunningly well on many difficult tasks. The power of GPT-3 relies on its extremely large 178 billion parameters and 45TB training data, also including different deep learning methods like unsupervised learning, zero-shot learning, fine-tuning, etc. There is not much innovation in terms of the model architecture, GPT-3 and its predecessors [3] all use a transformer-based neural network. However, training with such a large number of parameters and data can be quite challenging, not only because it requires a significant amount of computational resources and can be time-consuming, but also because there is a risk of overfitting, which occurs when a model becomes too closely attuned to the training data, resulting poor generalization to new data. GPT-3 shows that the transformer model is effective in addressing the overfitting problem and is able to scale up. The transformer model was first introduced by Google in 2017 [4]. It is a deep learning model based entirely on the self-attention mechanism, which is effective at capturing long-distance relationships or patterns in data by assuming that all input tokens are equidistant from one another. Compared with the previous popular models such as RNNs (recurrent neural networks), this approach allows for greater parallelism and a higher level of complexity, resulting in improved accuracy, precision, and speed. Since its introduction, the application of the transformer has rapidly expanded from neutral language processing to other areas like computer vision and audio processing, and has consistently outperformed traditional approaches. Its versatility and effectiveness make it a milestone in deep learning.

As the transformer has gained extensive popularity, more and more hardware re-


d

Please provide me with a guide to holidaying in Tahiti

Tahiti is a beautiful and unique destination for a holiday. Here are some tips to help you plan your trip:

1. When to go: Tahiti has a tropical climate with warm temperatures year-round. The dry season is from May to October, while the wet season is from November to April. Many people prefer to visit during the dry season, but the wet season can also be a good time to visit as the temperatures are cooler and the islands are less crowded.
2. How to get there: The main international airport in Tahiti is Papeete's Faa'a International Airport, which is served by a number of airlines including Air Tahiti Nui, Air France, and Hawaiian Airlines.
3. Where to stay: There are a variety of accommodation options in Tahiti, including hotels, resorts, and vacation rentals. Some popular areas for tourists to stay include Papeete, the island of Moorea, and the Bora Bora lagoon.
4. What to do: There are plenty of activities to enjoy in Tahiti, including snorkeling, diving, hiking, and visiting cultural sites. You can also relax on the beautiful beaches, go fishing, or take a boat tour of the islands.
5. Food and drink: Tahitian cuisine is a blend of French and Polynesian influences, and features dishes such as poisson cru (raw fish marinated in coconut milk), chicken fafa (chicken cooked in a taro leaf and coconut milk sauce), and po'e (a banana and taro pudding). Traditional drinks include coconut water and noni juice.

I hope these tips help you plan your holiday in Tahiti! Let me know if you have any other questions.

Figure 1.1: A dialog with ChatGPT. It accurately understands the question and gives a very comprehensive answer. (snapshot from the ChatGPT website)

searchers have turned their attention to ways of optimizing the performance of the model. Specifically, there has been great interest in finding ways to accelerate the inference and training for transformers. One approach is to design a domain-specific processor where all the efforts are focused on optimization for the unique requirements of transformers. This approach can achieve the best results in terms of performance, power, and area (PPA), but it also has the following drawbacks:

1. As the deep learning model evolves over time, the model the hardware specialized for could soon be outdated. Thus the domain-specific processor is more likely to be obsoleted and has a limited lifespan.

2. The domain-specific processor lacks flexibility, as it is only able to perform a specific set of tasks efficiently. This means that its potential market is small.

Since transformers provide more parallelism than previous models, it has opened up the possibility for general-purpose processors to achieve a better balance between performance and flexibility. This is because general-purpose processors are able to take advantage of the increased parallelism to perform transformer-based models more efficiently, making them a more viable option for a broader range of tasks. The vector processor is one of them that is designed to operate on vector data, which means a single instruction is applied to all the elements in a vector (SIMD), and it is particularly well-suited to tasks that require a large amount of data and high level of parallelism. Vector processors have been widely used in supercomputers which are designed for heavy engineering and scientific tasks, such as weather simulation and pharmaceutical research. The first vector machine was developed in 1976 by Cray Inc. with the introduction of the Cray-1. Since then, various vector architectures have been developed by different companies, including ARM's Scalable Vector Extension (SVE) and NEON, Intel's Advanced Vector Extension (AVX), and the RISC-V Vector instruction set (RVV). We are particularly interested in RVV due to its open-source nature and the opportunities for customization. At the Integrated Systems Laboratory at ETH Zurich, we have developed our own vector machine called Ara, which is compliant with the RVV 1.0 specification. Ara is equipped with 32 vector registers, and it is configurable with different numbers of lanes, which are independent execution units that are capable of performing tasks concurrently. Since there is no prior research on implementing transformer models on a vector processor, it would be highly interesting to examine the potential benefits and challenges of using this combination, particularly in the context of our own vector processor, Ara.

1.1 Objectives

The main objective of this project is to run the transformer model on our vector machine Ara, identify any performance issues or bottlenecks during execution, and implement various optimization strategies at both the hardware and software levels in order to improve the overall efficiency. Since Ara is designed to be energy-efficient and lightweight, our focus in this project will be on model inference rather than training. Because the training process usually requires much more computation power and energy, which is beyond Ara's capabilities. Additionally, the primary purpose of this project is to serve as a proof of concept. Therefore, we will focus on using a general model with the transformer architecture rather than focusing on a specific model.

1.2 Thesis Organization

The organization of this thesis is as follows: Chapter 2 presents background knowledge on the transformer model, AI accelerator, vector processor, and relevant prior research. Chapter 3 outlines the vectorization of the transformer model and analyzes the performance results. Chapter 4 discusses various hardware and software optimization methods based on the findings from Chapter 3. Chapter 5 compares the baseline and optimized results, and also the physical design resutls of the modified Ara. Finally, Chapter 6 concludes the thesis and offers insights on potential future work.

Chapter 2

Background and Related Work

2.1 Transformer Neural Network

2.1.1 Neutral Language Processing

NLP refers to the technology that enables machines to comprehend and generate text using human language and reasoning, with the goal of enabling machines to communicate with humans more naturally and intuitively. In addition to the chatbot we discussed before, NLP can also be applied to the following areas:

1. Sentiment Analysis, which involves using a machine to determine whether the input text conveys a positive or negative emotion, or even optimism or pessimism. For example, many quantitative trading companies use this technology to analyze news related to a particular stock in order to predict its future trend.

2. Speech Recognition, which is a technology that lets the machine interpret spoken language. With the deep learning approach, the voice is first converted into a machine-understandable signal called spectrogram, and a deep learning model is then used to extract information from this signal and generate human-like responses. This technology can be found in various devices and systems, such as digital assistants on smartphones like Siri.

3. Machine Translation uses a machine to translate a text from one language to another. It has undergone extensive development over time and its accuracy is now comparable to that of human translators. This has led many people to believe that professional translators may be at risk of losing their jobs in the future.

The deep learning approach has become highly successful in recent years due to its ability to achieve results that traditional machine learning algorithms cannot, which often require numerous complex pre-defined rules. The philosophy behind deep learning is to create an artificial neural network by mimicking the human brain's structure, which consists of billions of neurons connected in a complex network. By doing so, the deep learning model is able to extract and learn from the underlying information contained in the training data.

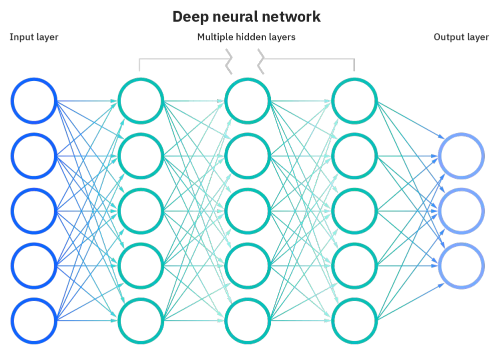


Figure 2.1: A simple deep learning network. Each neuron receives input from all the neurons in the previous layer and sends its output to the neurons in the subsequent layer. These layers can be numerous, hence the term "deep". (IBM Cloud Learn Hub/What are Neural Networks[5])

In order to utilize an NLP deep learning model, the input text must first be converted into a mathematical representation through a process known as word embedding. This involves projecting each token (e.g., word or phrase) in the text onto a high-dimensional vector space, such that vectors of words with similar meanings are close to each other and those corresponding to unrelated words are farther apart. This projection may also be obtained through another deep learning model.

2.1.2 Recurrent Neural Network

Prior to the emergence of transformers, RNNs were the dominant approach for NLP tasks. For the basic deep learning model, the information flow is unidirectional, meaning that the network cannot consider information from previous input when processing later input. This is a problem for NLP tasks, as the meaning of words can depend on the context in which they are used. For example, consider the sentence "I am a student at ETH." Without considering the previous word "student," the network may interpret it as a cryptocurrency rather than a university.

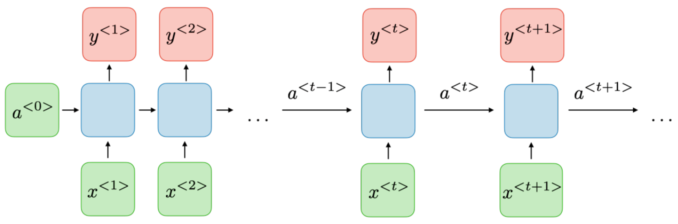


Figure 2.2: The structure of RNN. "a" represents the "activation" or the neuron state, 'x' represents the input to the neuron, and 'y' represents the output from the neuron. (Stanford CS 230 Recurrent Neural Networks cheatsheet[6])

Figure 2.2 illustrates the basic principle of a RNN. At each time step, the neuron state is a combination of the current input information x(t) and the previous state a(t-1). This allows the RNN to track sequential information with a very long length, as the hidden state at each time step depends on the hidden state at all previous time steps.

However, one of the main challenges with RNNs is the so-called "vanishing gradient" problem. During the backpropagation process, the gradient can become very small, especially for time steps that are far in the past. This is due to the chain rule, which states that the gradient of a composite function is the product of the gradients of its individual parts. As a result, the model weight updates may be nearly unaffected by the long-term time series, which means that the model may struggle to capture long-term dependencies in the data.

To mitigate this problem, the Long Short Term Memory (LSTM) RNN was proposed. LSTMs use a cell state to store long-term information, which can be read, erased, and written like RAM, allowing the model to better preserve dependencies over long sequences. While LSTMs do improve the ability of RNNs to learn long-distance dependencies, it is still possible for the vanishing gradient problem to occur, especially when training on very long sequences.

2.1.3 Transformer and Self-Attention Mechanism

The transformer model has revolutionized the field of NLP with its parallelizability and scalability. Figure 2.3 shows its architecture consisting of an encoder block and a decoder block, which work together to perform tasks like machine translation. The encoder processes the input sequence and converts it into a representation that the decoder can use to generate the translated output. In some applications, only one part of the transformer is used, such as when performing sentiment analysis, where the encoder alone is used.

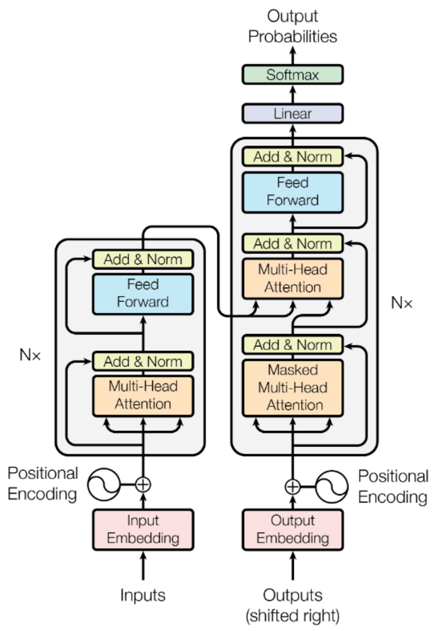


Figure 2.3: The transformer model architecture (Attention is all you need[4])

$$\begin{aligned} Q_i &= x \times W_{q_i} + b_{q_i} \\ K_i &= x \times W_{k_i} + b_{k_i} \\ V_i &= x \times W_{v_i} + b_{v_i} \end{aligned} \tag{2.1}$$

$$score_i = Softmax(\frac{Q_i \times K_i^T}{scale}); \; attention_i = score_i \times V_i \tag{2.2}$$

$$multihead\_attention = [attention_0, attention_1, ..., attention_{h-1}] \times W_o + b_o \tag{2.3}$$

The encoder and decoder have a similar structure, consisting of multiple blocks stacked on top of each other. Each block is composed of two layers: a multi-head attention layer and a feed forward layer. The decoder has one more multihead attention layer where the information from the encoder is used. The core part of the transformer is the self-attention mechanism used in the multihead attention layer. We can understand the concept of "self-attention" with an example of information retrieval: Imagine searching for a product on an e-commerce platform. The content we enter into the search engine is the Query, and the platform matches the Key (e.g., product type, functionality) relevant to the Query. The platform then retrieves matching content (the Value) based on the similarity between the Query and the Key. The Q, K, and V matrices in self-attention perform the same function as previously described. They are derived by multiplying the input matrix with their respective weight matrices and adding the bias term. In matrix computation, cosine similarity is a method used to calculate the similarity between two matrices. This similarity is represented by the attention score, which is calculated using the term $Q_i K_i^T$ in equation 2.2. The attention score is then passed through a Softmax layer and scaled for normalization. The output of the self-attention for a single head is obtained by calculating a weighted sum of V, using the normalized score as the weights. Multiple independent self-attention heads are performed parallel and merged to form the multihead attention. Essentially, each head attends to a different part of the input. For instance, when the model is trying to understand a portrait, one head may focus on the person in the image while another head focuses on the background. This allows the model to effectively process and extract information from different aspects of the input simultaneously. In order to introduce nonlinearity to the model, a feed forward layer is often added after the multihead attention. This feed forward layer consists of two linear transformations with an activation function applied between them. Additionally, to improve the training process and prevent the model from getting stuck in local optima, various connection layers are often used between sublayers. These include the residual connection layer, layer normalization layer, and dropout layer. Until now, the model cannot take into account the sequential order of the input data. To incorporate this information, a positional encoding is added to the input embedding to give the model a sense of the relative position of each token in the sequence. There are several ways to represent the position of each token, including using sinusoidal functions to represent periodic relative positions or learning an absolute position representation.

Transformer in Computer Vision

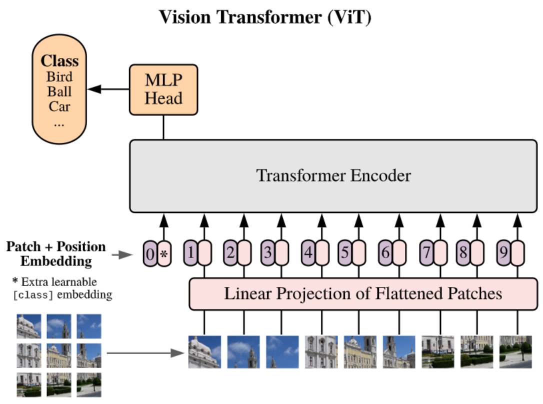

Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0*
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

Figure 2.4: ViT model overview: Splitting an image into a sequence of small patches, and then feeding the sequence to the transformer encoder. (An Image is Worth 16x16 Words[7])

Inspired by the successful application of the transformer in NLP, the Vision Transformer (ViT) [7] uses the transformer architecture with little modifications to replace the CNN to perform image classification tasks, and it can meet or exceed the state-of-the-art (SOTA) level. Unlike sequential language information, images are multi-dimensional, so the algorithm first needs to convert the image into a sequence as shown in figure 2.4. For example, an image with dimension $H \times W \times C$ (height, width, channels) can be split into smaller patches with dimensions $P \times P \times C$, where N, the number of patches, equals $H * W / P^2$, which is the sequence length. Then, a linear transformation is applied to each patch to suppress its dimensionality from $P^2 \times C$ to a single dimension D. This allows the model to analyze the image by considering the patches as a sequence of individual elements.

Another approach is to use a hybrid architecture such as the Swin Transformer [8] and MobileViT [9], which maintain the structure of a CNN but incorporate transformer layers as a complement. The transformer model is not well-equipped to handle high-resolution images due to their long sequence lengths. This is because the self-attention mechanism in the transformer has a computation complexity that is quadratic in the sequence length, making it expensive to process long sequences. On the other hand, CNNs excel at extracting local features and can reduce the sequence length of the input to the transformer. By combining the transformer and CNN models, it is possible to leverage the strengths of both approaches and improve the model's ability to learn both local and long-range dependencies in the data.

Challenges of Transformer

One of the main challenges of the transformer model is its computation complexity in self-attention, which grows quadratically with the sequence length. To address this issue, researchers have proposed various methods to reduce the complexity of the self-attention mechanism. The Linformer model [10] uses projections to map the Value and Key matrices to a lower dimensional space, such that the dimensionality changes from $n \times d_m$ to $k \times d_m$, where k is smaller than n. The Big Bird model [11] and Sparse Transformer model [12] exploit sparsity in the attention matrices by selectively skipping certain elements, resulting in a complexity of $O(n)$ or $O(n\sqrt{n})$ instead of $O(n^2)$. The Linear Transformer [13] offers another perspective. It views the self-attention as a kernel feature map and replaces the Softmax function with another kernel function that can utilize the associativity law of matrix multiplication. By doing so, the model can first compute $\phi(K^T) * V$, where ϕ is the new feature map, with a complexity of $O(n)$.

2.2 Transformer Accelerator

Deep learning algorithms have a profound impact on hardware development because they often require a large amount of data and computational resources. One of the characteristics of DNN is that most of the operations during the training and inference are Multiply-Accumulate operations ($o = a * b + c$). These MAC operations require four times memory access, which can lead to a significant memory bottleneck for modern CPUs since they require data to be transferred back and forth between the ALU and memory. Memory access is far more expansive than computation. For example, a study

[14] shows that in 45nm technology, a single DRAM read operation can consume 700x more energy than a 32b FP multiplication. To achieve high performance and low energy consumption in DNN, many novel hardware architectures have been proposed. These architectures often introduce innovations in the memory hierarchy, custom assembly instruction, and parallel processing. There are several metrics to evaluate that specialized hardware according to the book [15]:

1. **Accuracy**: The accuracy of a model can be defined in different ways depending on the task. For example, in the machine translation task, the BLUE score is used to evaluate the similarity between the translated text and the reference text. On the hardware side, one important factor that can impact accuracy is data precision (8b, 16b ...). Reducing the data precision can benefit the throughput and power, but it can also have a negative effect on the accuracy. One way to balance this trade-off is to use mixed precision, where low precision is used for some parts of the model, which are less sensitive to accuracy degradation.

2. **Throughput**: Intuitively speaking, the throughput can be thought of as how fast the execution is. To improve throughput, we can either increase the clock frequency to speed up all Processing Elements (PEs) or increase the number of PEs. However, boosting the clock is often limited by the physical implementation technology. And adding more PEs can lead to underutilization due to imbalanced workload or memory bottleneck.

3. **Energy Efficiency**: Power is crucial, especially for edge devices operating on battery power. High power consumption can lead to short battery life and overheating, which can damage the chip and slow down the transistors. Many techniques can be used in VLSI design to improve power consumption, such as clock gating, power gating, and using transistors with high threshold voltage (HVT). In the context of DNNs, power consumption is dominated by memory access, particularly access to off-chip DRAM. So reducing memory access times is very important. This can be done by maximizing the reusability of the data, exploiting the sparsity of the data (many DNN's parameters are zero-valued), etc. Additionally, these methods can also improve the throughput, as memory access is also time-costly.

4. **Hardware Cost**: There are many factors that can influence the cost of a chip, with the most important being the chip area. In general, the larger the area, the more expansive it is for tape-out. Memory also plays a major role here, as it usually occupies a large area. However, having a large on-chip SRAM can reduce off-chip memory access through caching, so it is again a trade-off between PPA.

5. **Flexibility and Scalability**: A good accelerator should be able to support a wide range of DNN models, as the field of DNNs is constantly evolving and new models are frequently being developed. An accelerator that can only perform efficiently on a specific DNN model may quickly become obsolete as the market shifts towards newer models, so it is important for an accelerator to be flexible and adaptable in

order to remain competitive. In addition, the accelerator should be scalable, which means it is able to increase its processing power by adding more PEs or combining multiple chips into one package (chiplet), while still maintaining high throughput. This allows the accelerator to meet the growing computation demand of DNN without the need to redesign a new chip.

The transformer model heavily relies on matrix multiplication (MatMul) as its main computational operation, making MatMul optimization the primary focus for a dedicated accelerator. In the following sections, we will examine various accelerator designs, including both general-purpose accelerators that can be applied to different models and specific accelerators that are exclusively for the transformer.

2.2.1 TPU and Systolic Array

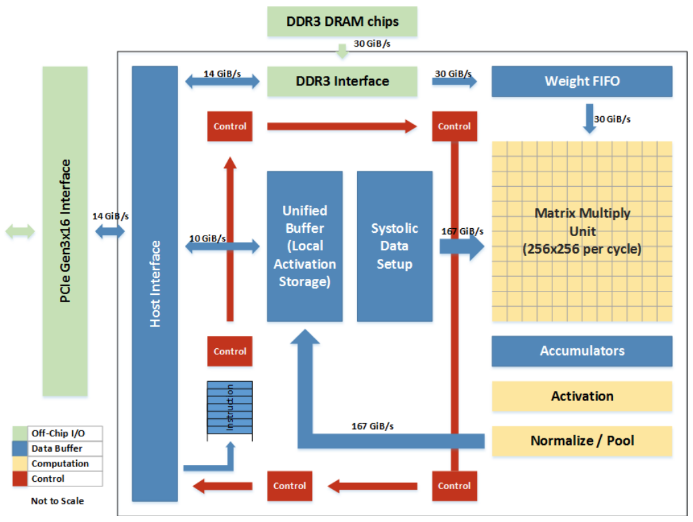


Figure 2.5: Block diagram of TPU (In-datacenter Performance Analysis of a Tensor Processing Unit[16])

A well-known and massively deployed DNN accelerator is Google's TPU[16]. Over several generations, TPU has become a very important computing infrastructure for Google's Cloud Services. Figure 2.5 shows the architecture of TPU. The core computation unit of TPU is the Matrix Multiply Unit (MMU), which is based on a systolic array architecture[17]. This MMU is a 256x256 array, with each element being a simple MAC PE. Because each PE is designed only to perform MAC operations, it is small in size, allowing for a large number of PEs to be included in the TPU. Specifically, the TPU has 64k PEs, offering a peak performance of 92 Tera Operations Per Second (TOPS). Located beneath the MMU, there are an activation unit and a normalization unit that apply functions such as ReLU or Sigmoid to the output of the MMU and store the results back in the Unified Buffer.

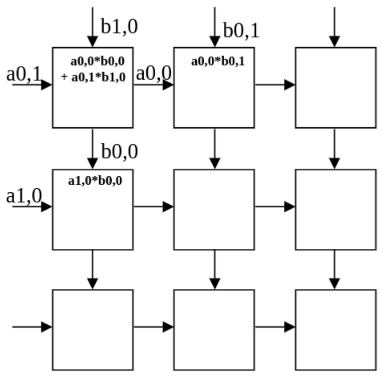


Figure 2.6: The second step of MatMul with a systolic array. (EECC756 Lecture 1, by Dr. Shaaban[18])

To understand the inner workings of the MMU, let's consider an example of matrix multiplication shown in figure 2.6. Assume B represents the weight matrix and is stored in the Weight FIFO, while A represents the activation matrix and is stored in the Unified Buffer. In the first step, PE(0,0) receives $a(0,0)$ and $b(0,0)$ and calculates the product $a(0,0) * b(0,0)$. It also passes $a(0,0)$ to its right neighbor PE(0,1) and $b(0,0)$ to its bottom neighbor PE(1,0). In the next step, PE(0,0) receives new data $b(1,0)$ and $a(0,1)$ and computes the product while adding it to the previous result. At the same time, PE(0,1) also receives $b(0,1)$ and multiplies it with $a(0,0)$ that it received from PE(0,0) in the previous step. By reading elements from both A and B only once from memory and utilizing them across all PEs in the same row or column, the demand for high memory bandwidth is reduced, leading to improved performance and power efficiency.

TPU is engineered to accelerate various DNN models, including CNN and RNN, as MMU is also able to execute convolution efficiently. However, the systolic array is a highly favored architecture for other accelerator designs as well. The transformer accelerator in this work [19] also employs a systolic array as the core computation unit. By optimizing the computation flow, it attains a high utilization rate of the systolic array.

2.2.2 CIM

The memory access bottleneck, also known as "the memory wall", is becoming increasingly critical in the post-Moore era. This is because when transistors get smaller and faster, memory technology has a much slower pace. To overcome the memory access bottleneck, various memory technologies such as High Bandwidth Memory (HBM), 3D-stacked memory, and non-volatile memory can be utilized. Additionally, a new architecture called Computing In-Memory (CIM) has been proposed to avoid data movement by directly processing the data in memory.

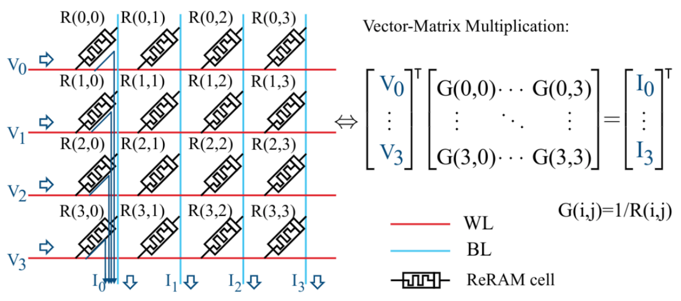

R(0,0)
R(0,1)
R(0,2)
R(0,3)
R(1,0)
R(1,1)
R(1,2)
R(1,3)
R(2,0)
R(2,1)
R(2,2)
R(2,3)
R(3,0)
R(3,1)
R(3,2)
R(3,3)
V_0
V_1
V_2
V_3
I_0
I_1
I_2
I_3
Vector-Matrix Multiplication:
$\Leftrightarrow \begin{bmatrix} V_0 \\ \vdots \\ V_3 \end{bmatrix}^T \begin{bmatrix} G(0,0) \cdots G(0,3) \\ \vdots \quad \ddots \quad \vdots \\ G(3,0) \cdots G(3,3) \end{bmatrix} = \begin{bmatrix} I_0 \\ \vdots \\ I_3 \end{bmatrix}^T$
G(i,j)=1/R(i,j)
WL
BL
ReRAM cell

Figure 2.7: Vector-Matrix multiplication with a ReRAM based crossbar array (ReTransformer[20])

The principle of the CIM-based accelerator is Kirchhoff's current law, which states that the sum of all currents flowing into a node equals the sum of all input currents flowing out. This can be applied to MatMul as shown in figure 2.7, where the output value (current I) is calculated by summing the product of the input value (voltage V) and weight (conductance R/G) at each cell, all in just one cycle. However, the crossbar used in this method is not well suited for other complex operations like normalization. Therefore, additional units, such as a vector unit or SIMD unit, are often employed to handle these operations. Research [20] [21] show that using a CIM-based accelerator for the transformer model can achieve a good balance between performance and power.

2.3 RISC-V Vector Processor

Vector processors, as a type of SIMD processors, can use a single instruction to process multiple data simultaneously, in a pipelined manner. This allows them to achieve high performance and throughput when working with a large amount of data. One key feature of vector processors is the ability to vary the vector length, which improves flexibility and efficiency in adapting to different workloads. The RISC-V Vector Extension (RVV) is an open-source instruction set that defines the functionality of vector instructions, like vector load/store, vector arithmetic operations.

The Ara SoC [22] [23] is a RISC-V processor that adheres to the RVV 1.0 specification, providing support for both vector and scalar operations. The Ara co-processor works in conjunction with a RISC-V scalar core CVA6, which forwards all vector instructions to Ara after preliminary decoding. Figure 2.8 is a block diagram of Ara, illustrating various components and their interactions. Upon receiving a request from CVA6, the instruction is first sent to the dispatcher for decoding. Here, the dispatcher also updates the CSRs in case of configuration instructions, like vector length or data format settings. Through this process, the dispatcher identifies the operation to be executed, the vector registers to be used, and the number of elements to be processed. This information will be transmitted to the main sequencer for issuing and hazard tracking. The sequencer first determines the functional unit where the instruction should be directed and checks if there is a structure hazard. Furthermore, it also records the usage of vector registers by each active instruction in order to examine different types of data hazards. The Vector Load/Store Unit (VLSU) is responsible for data loading and storing. It has a connection to the off-chip memory via an AXI protocol. The ADDRgen, which is an integral part of the VLSU, generates the memory addresses in accordance with the specified base address and memory access pattern. Subsequently, the Vector Load Unit (VLDU) and Vector Store Unit (VSTU) take over, executing the task of writing or reading the data to and from the vector register file (VRF), respectively. The Mask Vector, an essential element in the system, is used to selectively skip the execution of certain elements within a vector. To achieve this, the system incorporates the Mask Unit (MASKU) which transmits the mask data to the appropriate lanes. In addition, the MASKU is also equipped to execute operations that specifically operate on mask data, such as the vfirst instruction which returns the index of the first occurrence of a true value within a mask vector. In certain situations, we may want to manipulate the order of elements within a vector, such as shifting them to the right. Since the vector data is shuffled among the lanes, we need a slide unit (SLDU) as a transfer station. The SLDU is connected to all the lanes and is able to move data from one lane to another.

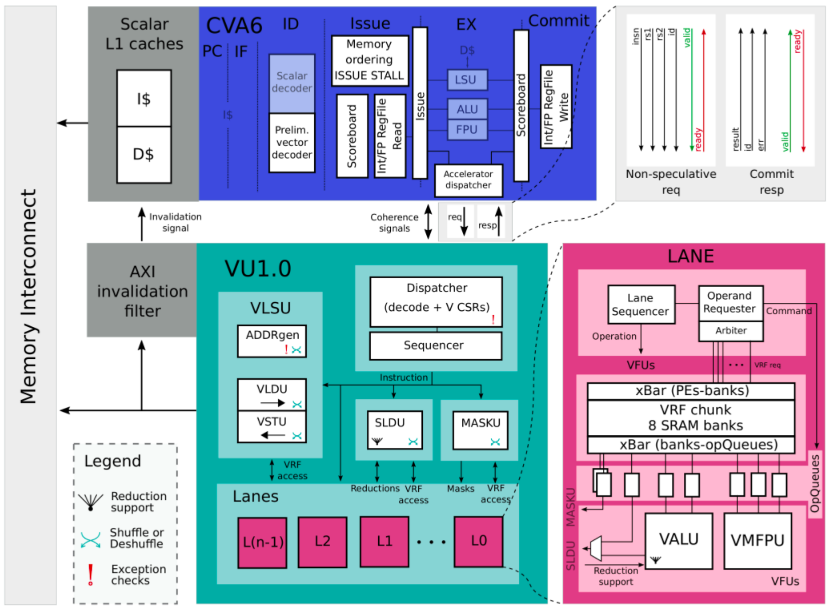


Figure 2.8: The top-level block diagram of Ara SoC (A "New Ara" for Vector Computing[23])

All the remaining vector instructions are carried out by the lanes, where the number of lanes is configurable. Each lane has its own VRF, which stores a portion of the entire vector registers (consisting of 32 registers). To ensure an even distribution of workload among lanes, vector elements are stored in a shuffled manner, allowing each lane to process a similar number of elements. The lane sequencer further refines the requests it gets from the main sequencer to determine which sub-units in the lane need to be involved in the current operation. It instructs the operand requester to retrieve elements from the VRF, and the VRF subsequently sends elements to the corresponding operand queues. Depending on the data type, we have specialized units for handling integers (VALU) and floating-point numbers (VMFPU), respectively.

Chapter 3

Transformer Vectorization

3.1 Methodology

In this section, we will present various strategies and techniques used in this project.

- As previously discussed in the background section, the transformer model's encoder and decoder share a similar architecture, comprising of multiple identical blocks. In light of this, our research will concentrate on a single encoder sub-block, specifically the multihead attention combined with a feed forward layer.
- In adherence to the naming convention used in the original transformer paper, "Attention is all you need [4]", we refer to the second layer in an encoder sub-block as the "feed forward layer", which may also be referred to as a "fully connected layer (FC)" or "multi-layer perceptron (MLP)". The input sequence length is represented by n, corresponding to the number of rows in the input matrix. The embedding size, or the number of columns in the input matrix, is represented by *d_model*. *h* represents the number of heads in the multihead attention, and *dk* is the feature size of each head, which is equal to *d_model* divided by *h*. *d_ff* represents the feature size of the feed forward layer, which is typically four times the size of *d_model*.
- We adopt the model size configuration from the BERT model [24] and use three sizes: "base", "large", and "huge". Additionally, we also use four different sequence lengths. The specific parameter values are presented in tables 3.1 and 3.2.
- As the updated version of Ara does not yet support an operating system, we implement the transformer model using bare-metal C programming and test the performance through RTL simulation. Figure 3.1 illustrates the overall verification structure. We first implement a golden Python transformer to generate the golden result. This golden model has the same functionality as other actual transformer models, but is more friendly for debugging and troubleshooting. To ensure the correctness of the Python model, we use the annotated transformer [25] as a

	base	large	huge
d_model	768	1024	1280
h	12	16	16
dk	64	64	80
d_ff	3072	4096	5120

Table 3.1: Parameter values for different model sizes

n	64	128	256	512

Table 3.2: Different sequence lengths

reference. We copy the parameters of each layer from the annotated transformer and apply them to our python transformer, and then compare the output. Potential errors can originate from both the hardware and the C transformer implementation, so it is essential to identify and correct them as early as possible. To avoid costly and time-consuming RTL simulation, we first use a RISC-V ISA simulator called Spike [26] to verify our C transformer implementation. The Spike simulator is able to execute the RISC-V assembly code and provides debugging functionalities such as viewing the content of a memory address or register. This allows us to quickly identify and address any errors at the software level before proceeding with more in-depth hardware simulations.

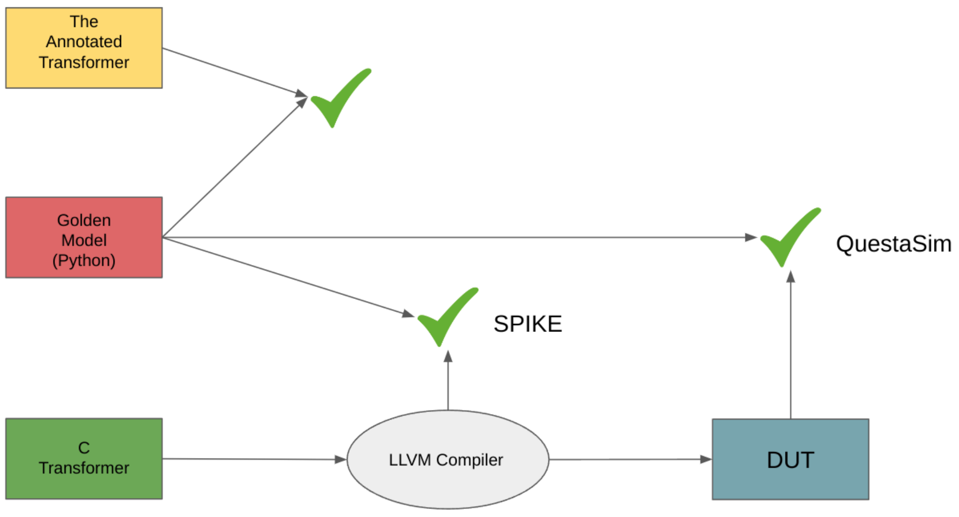


Figure 3.1: Multi-level verification

3.2 Transformer Benchmark

3.2.1 MatMul

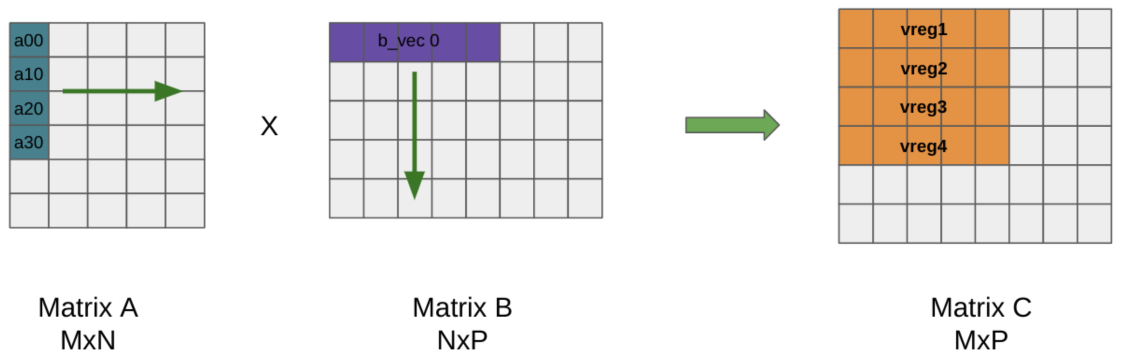


Figure 3.2: 4-slice MatMul, process four rows at a time

The fundamental operation in the transformer models is matrix multiplication (MatMul), which accounts for over 95% of the total execution time. It typically involves element-wise multiplication of a row of matrix A and a column of matrix B, followed by a reduction to sum all elements in the resulting vector. However, the reduction operation is less efficient than other vector instructions on Ara, as it cannot fully leverage parallelizability due to data dependency. To eliminate the use of reductions, we use a MAC-based matrix multiplication method as shown in Figure 5.2. This method involves loading several elements from a column of matrix A into scalar registers based on the number of slicing s. Then, one row of matrix B is loaded into a vector register. Using vector-scalar MAC (*vfmacc.vf*) operation, each scalar value in A is multiplied with b_vec and the intermediate results are stored in s number of different vector registers. By repeating this process along the shared dimension N, we obtain a sub-matrix of the final result matrix C. To further enhance efficiency, the optimized algorithm also employs "loop unrolling" by loading two rows of B and two columns of A in the inner loop, thus reducing the overhead of branch instructions. Additionally, to improve utilization of the FPU, we manually incorporate vector store instructions when the program reaches the N-th loop, resulting in immediate storage of results in memory after each completed row of computation, rather than waiting for all s rows to finish.

A bias term is often added to the resulting matrix in DNNs. Separately adding the bias results in additional data loading and storing, leading to increased energy consumption and low throughput. To address this, we have implemented a variant of matrix multiplication called "matmul_bias" that applies the bias before storing the result back in memory. Similarly, we also have "matmul_transpose" and "matmul_add", which transpose the result using vector strided store and add another matrix element-by-element, respectively.

3.2.2 Dropout and ReLU

Dropout and ReLU both set certain values to zero, but they serve different purposes. Dropout is used to prevent overfitting and make the model more robust by setting values to zero based on a probability p, and scaling the remaining values by 1/(1-p). Since random functions are not available on Ara, a selection mask is used instead. The Python golden model generates the selection mask based on the specified probability, and elements are set to zero on Ara only when the corresponding mask bit is 0. ReLU is an activation function that introduces non-linearity and sparsity to the model by setting all negative values to zero and maintaining positive values unchanged. This is accomplished by using a vector comparison instruction to identify and zero out negative values.

A shared characteristic of these two kernels is that the input matrix can be flattened and viewed as a 1D vector. Therefore, the vector length is not limited by the width or height, but by width * height. To increase the vector length further, we use the maximum length multiplier (LMUL8), which concatenates eight vector registers to form one register with 8x the length.

3.2.3 Softmax and LayerNorm

Softmax is a function that maps input values to a probability distribution in the range of [0,1]. In self-attention, it converts the similarity scores between the query (Q) and key (K) matrices into probabilities. This allows the attention mechanism to focus on the most important elements of the input. For classification tasks, Softmax is also commonly used as the output layer to predict the probability of each class. The calculation of Softmax involves three loops. First, find the maximum value in each row, then subtract it from each element, exponentiate the result ($exp(x_i - x_{max})$), and finally divide each value by the sum of all the values in the row. This ensures that the sum of all elements in the output is equal to 1, which represents a probability distribution.

$$Softmax(x_i) = \frac{exp(x_i - x_{max})}{\sum_j exp(x_j)} \tag{3.1}$$

Normalization is a technique in DNNs to make the model learn faster and converge better. Two popular forms of normalization are BatchNorm and LayerNorm. BatchNorm normalizes the values of a given dimension across all samples in a batch, while LayerNorm normalizes the values of all dimensions for a single sample, which is proved more effective for NLP data. The calculation of LayerNorm is shown in equations. We first iterate through the data twice to compute the mean and variance. Then the data is normalized by subtracting the mean and dividing by the variance (ϵ is used to prevent division by zero), resulting in a zero mean and unit variance distribution. As an optional step, the normalized data can be further scaled by applying weight (α) and bias (β) terms, which are learnable parameters.

1. $m = (\sum_i x_i)/d\_model$

2. $v = (\sum_i (x_i - m)^2)/(d\_model - 1)$[1]
3. $x'_i = ((x_i - m)/\sqrt{v + \epsilon}) * \alpha + \beta$

Since these two kernels both process the data within a row, like finding the maximum value in a row, using reductions can be quite inefficient. A solution to this is to vectorize over the columns rather than the row, such that each vector contains the same feature from different samples. To achieve this, we need to use strided memory operations, which automatically increase the memory address by the stride size after each element load or store. In this context, the stride size equals the number of elements in a row [2].

3.2.4 Self-Attention

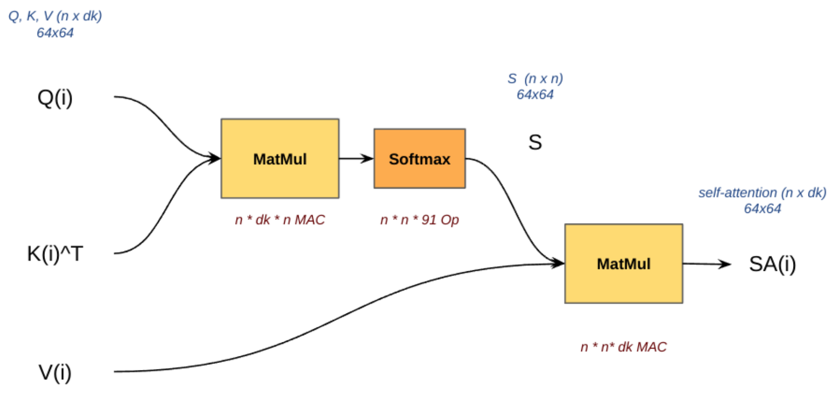


Figure 3.3: Self-Attention calculation of one head

The input of the self-attention layer is the Q, K^T, V matrices. K is pre-transposed using the "matmul_transpose" variant. The calculation flow is shown in figure 3.3, which includes two MatMul and one Softmax. To reduce the computational cost of vector divisions, the scaling $(Q * K^T/\sqrt{dk})$ is performed by scaling the weight and bias of Q (W_q and b_q) in advance, as shown in equations 3.2 to 3.4.

$$W'_q = W_q/\sqrt{dk}, \quad b'_q = b_q/\sqrt{dk} \tag{3.2}$$

$$Q' = x \times W'_q + b'_q = Q/\sqrt{dk} \tag{3.3}$$

$$Q' \times K^T = (Q \times K^T)/\sqrt{dk} \tag{3.4}$$

Multiple heads should be concatenated to form a large matrix, having a separate concatenation step can lead to extra memory access. To avoid this, the second MatMul stores the result directly at the position after concatenation using a pre-determined base address, eliminating the need for additional loading and storing of matrices.

[1]Bessel's correction
[2]Stride size in bytes

3.2.5 Multihead Attention

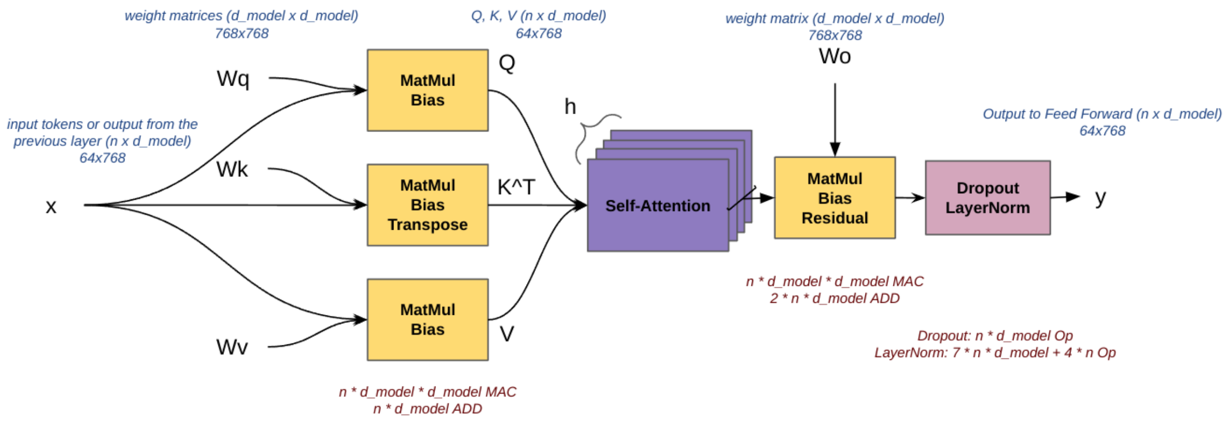

weight matrices (d_model x d_model)
768x768
Q, K, V (n x d_model)
64x768
weight matrix (d_model x d_model)
768x768
Wo
Wq
MatMul Bias
Q
h
input tokens or output from the previous layer (n x d_model)
64x768
Output to Feed Forward (n x d_model)
64x768
Wk
MatMul Bias Transpose
K^T
Self-Attention
MatMul Bias Residual
Dropout LayerNorm
y
x
n * d_model * d_model MAC
2 * n * d_model ADD
MatMul Bias
V
Wv
Dropout: n * d_model Op
LayerNorm: 7 * n * d_model + 4 * n Op
n * d_model * d_model MAC
n * d_model ADD

Figure 3.4: Multihead Attention, 1) calculate Q,K,V of all heads 2) self-attention 3) MatMul+Residual+Dropout+LayerNorm

Multihead Attention can be broken down into three steps. First, calculate the Q, K, V matrices. In principle, all heads can run in parallel and independently, but calculating them individually can lead to low hardware utilization. As shown in Figure A, if we multiply the input x with a single head h1 of shape $d\_model \times dk$, the vector length is dk (64 for a base-size model), and each lane can only acquire dk/nr_lanes number of data. However, if we merge the weights of all heads together, we obtain Q matrices of all heads horizontally concatenated. And the corresponding vector length is d_model, which is large enough for optimal hardware performance. In the second step, each head fetches its Q, K, and V matrices from the merged matrices using an offset and calculates the self-attention value. Finally, the concatenated matrix is passed through a linear layer with a residual connection, and subsequently, Dropout and LayerNorm are applied.

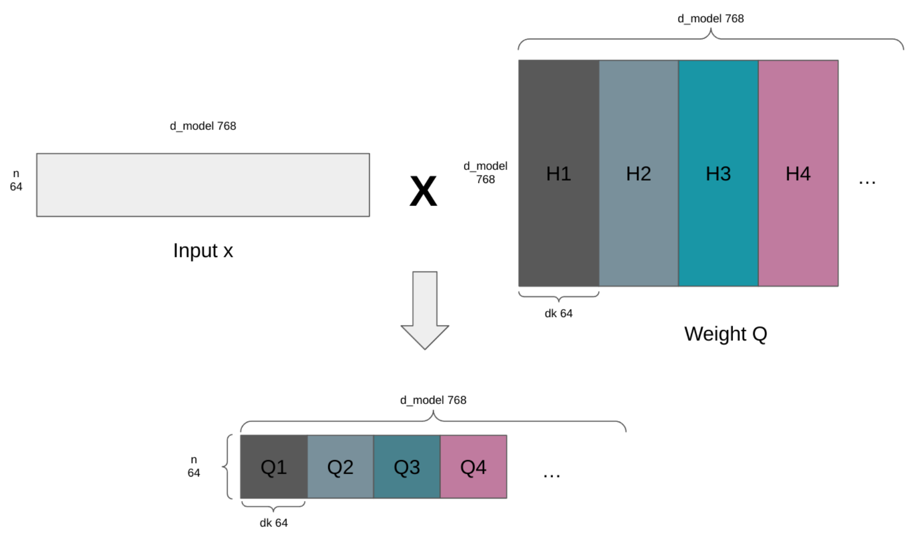


Figure 3.5: Improve Hardware performance by combining the weights of Q/K/V to form a large matrix.

3.2.6 Feed Forward

Feed Forward contains two MatMul and a ReLU activation in between. The first MatMul increases the feature size by a factor of four ($d\_ff = 4 * d\_model$), and the second MatMul reverts it to its original size. Dropout and layer normalization are applied afterward, similar to Multihead Attention.

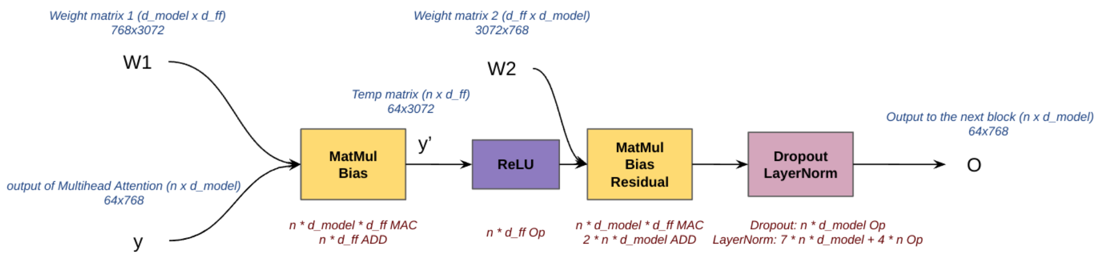


Figure 3.6: Feed Forward, a linear layer with normalization

3.3 Evaluation

Our simulation of the transformer benchmark includes both the Feed Forward and Multihead Attention sub-blocks, as well as all of the individual kernels within them. The results, as depicted in Figure A, provide a detailed breakdown of the execution times for the various components of the transformer model, with a base model size, a sequence length of 64, and 4 lanes. It is very obvious that the MatMul kernel is the primary contributor to the overall execution time, accounting for more than 95% of the total execution time. Conversely, all other kernels are relatively insignificant in terms of their negligible percentages. Feed Forward consumes almost double the execution time of the Multihead. This disparity can be explained by the difference in time complexity of the MatMul operation within the two sub-blocks. Specifically, the MatMul operation in Feed Forward has a time complexity of $O(n * d\_model * d\_ff)$, while for Multihead Attention, it is $O(n * d\_model * d\_model)$. This difference results in a fourfold reduction in time complexity for Multihead Attention as $d\_ff = 4 * d\_model$. As a result, two MatMul operations in Feed Forward are equivalent to eight MatMul operations in Multihead Attention. And since Multihead Attention has four MatMul operations in total (ignore the MatMul in self-attention), it has approximately half the execution time of Feed Forward. The execution time consumed by 12 self-attention heads occupies less than 0.3% of the total execution time. This is because the MatMul operation within these heads has an even smaller time complexity of $O(n * n * dk)$, which is 144 times less than $O(n * d\_model * d\_model)$ assuming $n = 64$ and $dk = d\_model/12 = 64$.

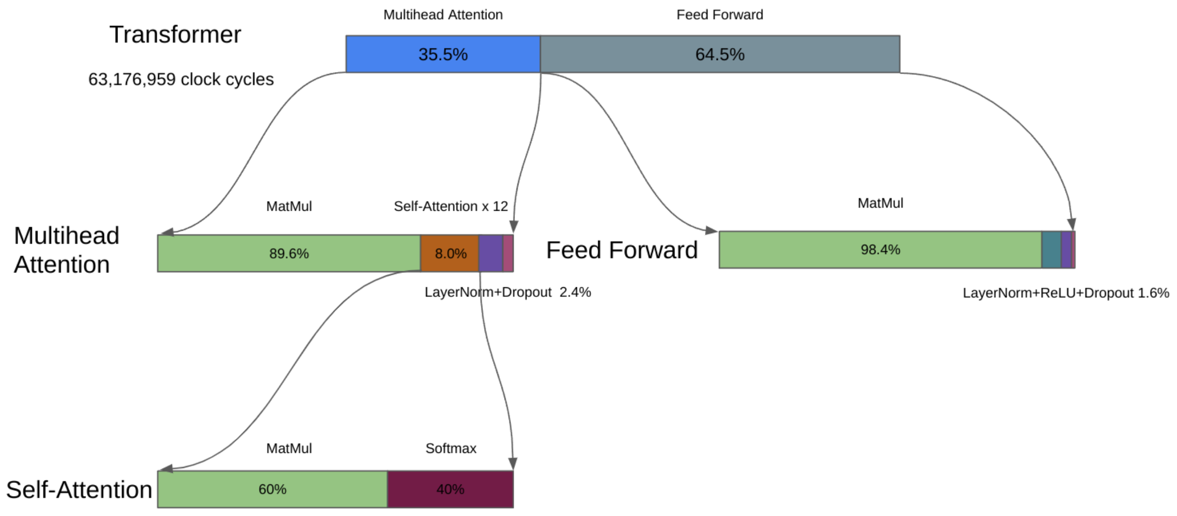


Figure 3.7: Execution time distribution

To identify potential bottlenecks and opportunities for optimization, we also examined the performance and FPU utilization of each kernel as presented in Chart 3.8. The performance is measured in Single-Precision Giga Floating-point Operations Per Second (SP-GFOPS), counting only the floating-point operations. The results indicate that the overall utilization of the transformer exceeds 90%, with an estimated performance of 14.59 SP-GFOPS (combining Multihead Attention and Feed Forward), indicating that the performance of the processor is almost fully utilized. Notably, for vector processors such as Ara, the vector length is a critical determinant of performance. In the case of the MatMul operations, which constitute the majority of the execution time, the utilization is particularly high as they are able to leverage the maximum vector length.

However, it is evident that other kernels exhibit inferior performance on Ara, particularly LayerNorm, which demonstrates a mere 8.52% utilization. Next, we will analyze and classify the underlying cause of this low utilization.

For Dropout and ReLU, both of which load the data once and perform a single operation, it is obvious that they are constrained by memory access. This implies that in most instances, the FPU is idling while waiting for data to be loaded.

LayerNorm and Softmax involve loading columns of matrices into vector registers, which poses two significant challenges. Firstly, the sequence length (n) used in this experiment, corresponding to the number of rows in the input matrix, is inadequate compared to the feature size (d_model). This causes a restricted vector length, thereby impeding the processor's performance. Secondly, using strided memory operations to load the columns of matrices leads to a substantial loss of performance. This is because strided memory operations necessitate sending one AXI request for every element, thus limiting the utilization of memory bandwidth to $1/nr\_lanes$. ($memory\ bandwidth = 32bits * nr\_lanes$). In addition, AXI requests are not consecutive but interspersed with at least one cycle stall between adjacent requests. So strided memory operations have longer memory access time as they cannot take advantage of the burst mode.

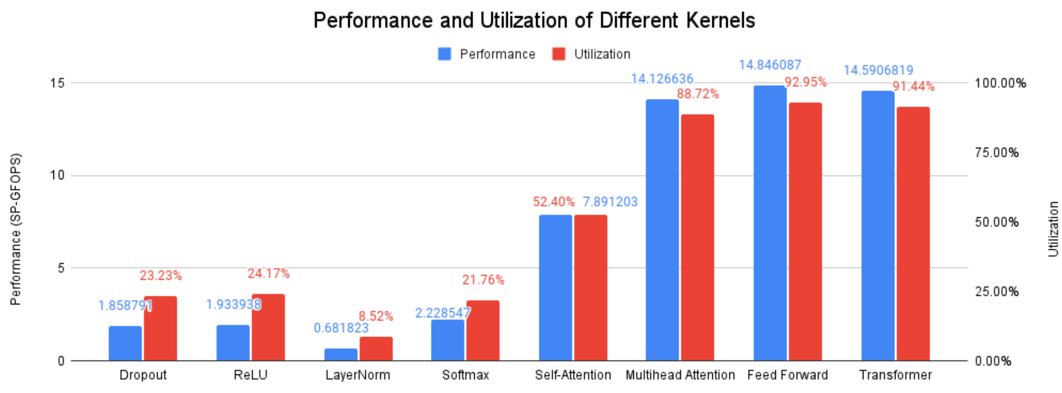


Figure 3.8: Performance and Utilization of Different kernels

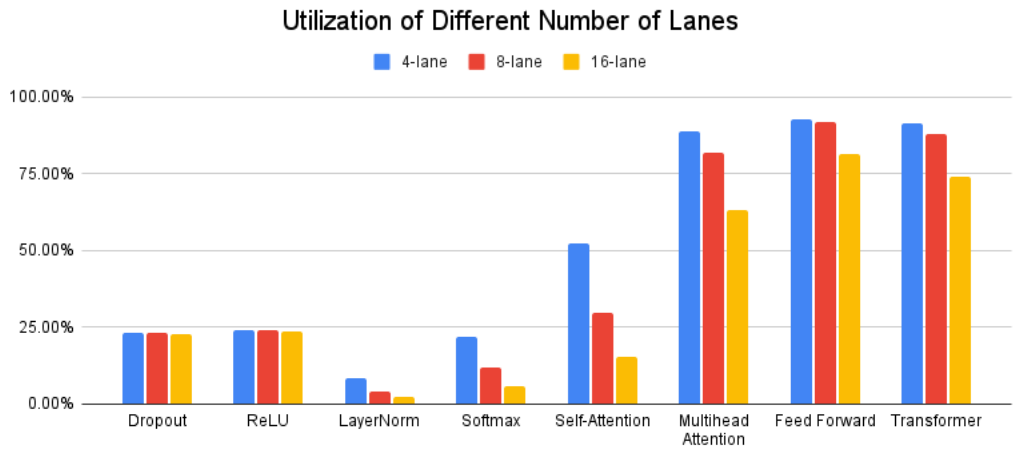


Figure 3.9: Utilization of Different Number of Lanes

Ara is a scalable system that can accommodate up to 16 lanes, but it is crucial to remember that adding more compute units does not necessarily guarantee an increase in performance. In order to gain a more comprehensive understanding of the system's performance, benchmarking was also conducted using 8 and 16 lanes. In this configuration, increasing the number of lanes also increases the maximum vector length, as each lane is equipped with its own VRF. For instance, $vlmax_{8lanes} = 2 * vlmax_{4lanes}$. As depicted in Chart 3.9, the utilization of Dropout and ReLU remains stable at slightly below 25%, indicating that both kernels can utilize the extra lanes. This is due to their ability to always supply the hardware with the maximum vector length by treating the input matrix as a long vector. On the other hand, LayerNorm and Softmax are unable to capitalize on the increased number of lanes, as their utilization decreases by half when the number of lanes is doubled. This is because more lanes require more data to maintain high utilization. The vector length of LayerNorm and Softmax is the sequence length (n), which is too short to effectively utilize the hardware. The same applies to self-attention, which is composed of two 64x64x64 MatMul operations and one Softmax operation. In addition to the decreased utilization of Softmax, MatMul also experiences a drop, as the vector length of 64 is insufficient to feed the hardware. To gain a deeper understanding of why utilization decreases when the number of lanes is increased but the vector length remains fixed, let's examine the example provided in the waveform 3.10. "inst1" is a vector instruction that operates on 64 32-bit elements. Since the data bus of Ara is 64-bit, in the case of 8 lanes, each lane must process 8 data, and in the case of 16 lanes, each lane must process 2 data. Additionally, Ara's FPU has latency, indicating that it takes multiple cycles for the FPU to produce results. Therefore, 8-lane Ara requires 9 clock cycles to complete "inst1", while 16-lane Ara requires 7 clock cycles, which is only 2 cycles less despite the doubled processing units. In theory, we could increase the instruction issue rate to keep the FPU pipeline full, but it is influenced by multiple factors such as the CVA6 scalar core requiring time to load the scalar operand from the cache (which can have an even longer latency when cache misses occur).

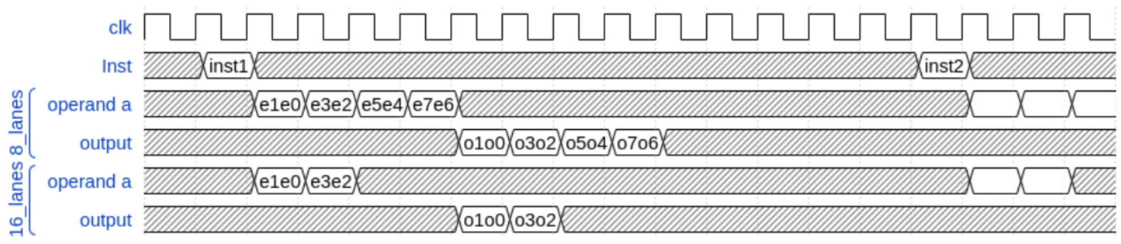


Figure 3.10: Comparison of execution of the same instruction in 8-lane and 16-lane configurations.

Multihead Attention and Feed Forward still retain high utilization due to their internally dominant large MatMul operations, which consistently allow for the use of the maximum vector length. An interesting point to note is that Multihead Attention experiences a significant decrease in utilization at 16 lanes, primarily attributed to an increase in the proportion of low-utilized self-attention from 8% to 20%.

Finally, we investigated the impact of sequence length (n) and model size (d_model) on performance, focusing specifically on LayerNorm, Softmax, and Self-Attention. For certain kernels, such as ReLu and large MatMul, increasing n or d_model does not affect the utilization, as it simply results in more iterations. This also holds true for the effect of d_model on LayerNorm, as the input shape is $n \times d\_model$ and n represents the vector length. For the second MatMul in Self-Attention, the vector length is dk, which remains constant for the base and large models. It increases to 80 when $d\_model = 1280$, but it has minimal impact on the overall utilization.

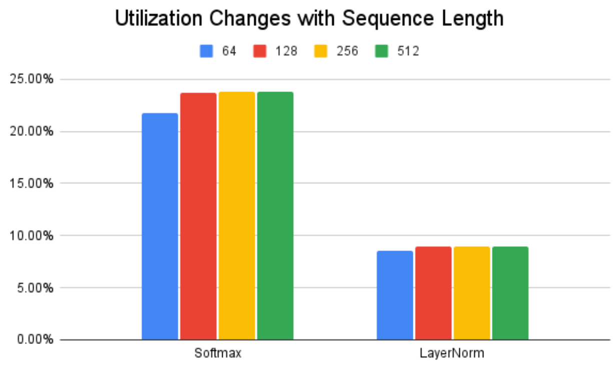


Figure 3.11: Utilization Changes with Sequence Length

The utilization changes in response to changes in sequence length are illustrated in Chart 3.11. As n represents the vector length for Softmax and LayerNorm, we would expect to see an increase in utilization as n increases. However, the utilization only increases slightly from n=64 to 128 and remains relatively constant thereafter. The limited utilization growth is largely attributed to the memory-bound nature of both kernels. Despite the large vector length, the duration of FPU operations is quite brief. Furthermore, the use of strided memory operations diminishes the advantages of a long vector.

Chapter 4

Software-Hardware Optimization

Our analysis of the transformer benchmark reveals that optimization efforts should target the underutilized LayerNorm, Softmax, and small MatMul, with a secondary focus on power optimization for large MatMul. We proposed a new MatMul algorithm along with custom vector instructions and hardware that is capable of efficiently handling both small and large matrices, as well as their transposed counterparts. Additionally, we enhance the performance of LayerNorm and Softmax by transposing all matrices in the transformer.

4.1 Software

4.1.1 Broadcast MatMul

Vanilla matmul has a vector length that is dependent on the number of columns in the second matrix operand. If this dimension is small, hardware utilization is low even if other dimensions are substantial. Furthermore, as the number of lanes increases, the requirement for a longer vector to sustain high hardware utilization arises due to the even distribution of vector elements across each lane. To address this issue, we developed a new MatMul algorithm that leverages broadcasting to consistently utilize the maximum vector length. The novel algorithm is rooted in the original, but differs in that it broadcasts the b-vector to each lane, thereby equitably furnishing each lane with an adequate amount of data for processing. Additionally, unlike the original algorithm that allows CVA6 to read the scalar elements of the A matrix and integrate them into instructions sent to Ara, the new algorithm reads the elements of the A matrix as vectors and stores them in the VRF. The new algorithm is demonstrated in Figure **??**. Initially, a column from the A matrix is loaded into the vector register. Ara distributes the elements of the vector evenly across each lane, providing each lane with multiple elements. Subsequently, a row from matrix B is loaded and broadcast to each lane. In every lane, the b-vector is multiplied by each element of the a-vector, with the outcome being stored in the VRF. This ensures that every lane receives a complete b-vector, thereby maximizing utilization, and reusing the b-vector multiple times, regardless of the number of lanes.

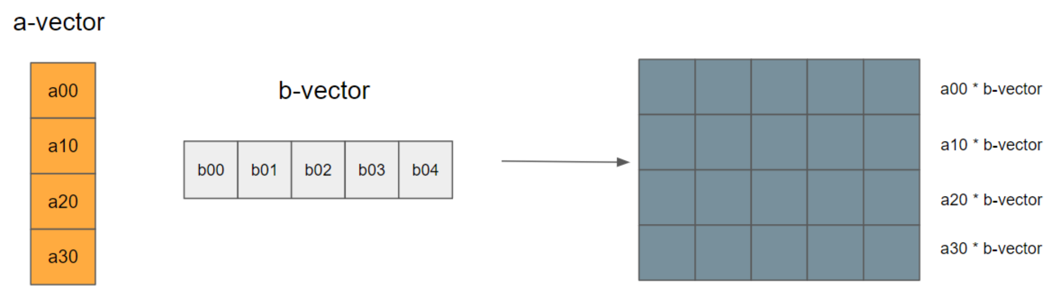
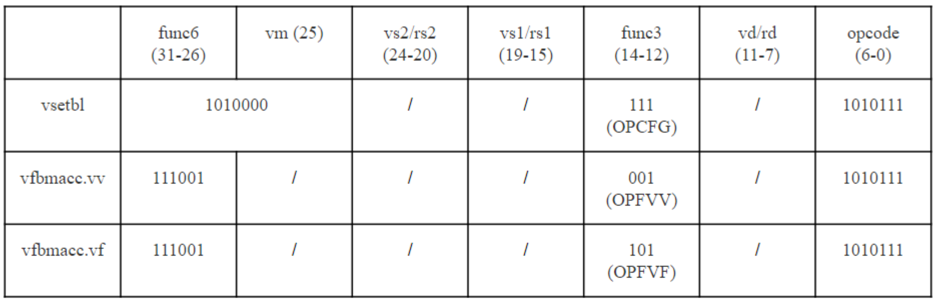
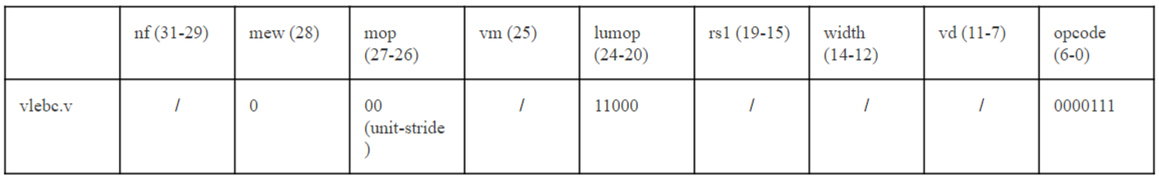


Figure 4.1:

To implement the new algorithm on Ara, we introduced a set of custom vector instructions. The first instruction is *vsetbl*, which sets the length of the broadcast vector based on the value stored in rs1. Then, using the broadcast vector loading instruction *vle<width>bc*, the b-vector is loaded into the system with the specified broadcast length and base address. The broadcast MAC instructions *vfbmacc.vv* multiplies the broadcast vector by each element in *vs1*, while adding the result to *vd* and storing it back in *vd*. To eliminate the need for initializing result vector registers to zero, we also have the *vfbmacc.vf* MAC operation that adds the value in rs1 (set to zero when performing MatMul) to the multiplication result instead of *vd*. Finally, a special strided store instruction *vsse<width>bc* is used to store the result from VRF to memory. Figure **??** displays the encoding of the new instructions. Instructions are encoded based on a similar instruction, with necessary modifications to differentiate them. For instance, the configuration instruction category encompasses both the broadcast length setting (*vsetbl*) and vector length setting (*vsetvl*) instructions. The *vsetbl* instruction is assigned a unique code in the 31-26 bit range, while retaining consistency in the 0-24 bit range with that of *vsetvl*. A special case is *vsse<width>bc*. As there is limited encoding space for strided store instructions, creating a new opcode would pose a significant challenge. To circumvent this problem, we use the same code for the strided store and broadcast strided store. The system will differentiate it as the new strided store instruction if it recognizes a non-zero broadcast length. In order to integrate the new instructions into the LLVM compiler, we modified the backend of LLVM. The frontend of the compiler serves as an interface, transforming high-level source code into an intermediate representation. This representation is then optimized and forwarded to the backend for the generation of the target assembly code. The inclusion of support for the new instructions within LLVM is relatively simple because we only use the inline assembly code for the new instructions. The sole requirement is that the compiler can successfully parse the program and recognize the assembly instructions. This is achieved by defining the new instructions within the existing RISC-V library of the compiler and specifying the register needed for each instruction.



	func6 (31-26)	vm (25)	vs2/rs2 (24-20)	vs1/rs1 (19-15)	func3 (14-12)	vd/rd (11-7)	opcode (6-0)
vsetbl	1010000		/	/	111 (OPCFG)	/	1010111
vfbmacc.vv	111001	/	/	/	001 (OPFVV)	/	1010111
vfbmacc.vf	111001	/	/	/	101 (OPFVF)	/	1010111



	nf (31-29)	mew (28)	mop (27-26)	vm (25)	lumop (24-20)	rs1 (19-15)	width (14-12)	vd (11-7)	opcode (6-0)
vlebc.v	/	0	00 (unit-stride)	/	11000	/	/	/	0000111

Figure 4.2:

The optimized MatMul program can be found in the appendix, and a succinct overview is presented here. First of all, whether the A matrix is transposed or not has a minimal impact on the overall performance. If the A matrix is transposed, we can load the row vectors of A using unit-stride load and burst mode, which returns the best performance. If A is not transposed, column vectors are loaded using strided load. Although this incurs a longer delay in each element reaching the processing unit, the performance impact is negligible since each element has to be multiplied by the entire b-vector, leaving sufficient buffer time for the next element to arrive. The length of the a-vector is determined by *NR_LANES* and *REUSE_SIZE*, being their product. Each lane holds *REUSE_SIZE* elements from a-vector and each element uses the b-vector once. For 32-bit data, the maximum broadcast length of 32 (1024 bits) is selected. This is because, at the maximum vector length, each lane handles 1024-bit data. Any further increase in broadcast length would require $LMUL > 1$, resulting in a more complex design. Like *LMUL*, the new MAC instruction uses multiple vector registers implicitly. The A elements in each lane come from different rows, requiring corresponding results to be stored in separate vector registers. For instance, if *vd* is specified as v8 and *REUSE_SIZE=4*, then each lane uses $v8$, $v9$, $v10$, $v11$ to store the results of four rows. To simplify hardware design, explicit store instructions are assigned to each used vector register for result storage, instead of using a single complex store instruction to store results of all used vector registers. In addition, the new algorithm is more efficient if the result matrix is stored in transposed form due to the characteristics of Ara (as explained in the hardware section). A new store instruction will be required if the normal form is necessary. But coincidentally, as will be described in the next section, transpose matrices are well-suited for transformers and can greatly improve the performance of other kernels.

We also developed bias and residual versions of MatMul. However, since the result data produced by the new algorithm are not in the same row or column, but are distributed in different rows and columns, the third operand data must be processed to meet the format requirement. As those operations are minor, we opted to use another loop to load and store the matrix rather exploring further in this area.

4.1.2 Transpose the Transformer

Softmax and LayerNorm are limited by strided memory operations, but if the input matrix is transposed, they can benefit from unit-stride memory operations with burst mode. Thus, transposing the matrices in the transformer model is the second optimization point. We assume that the input embedding is already transposed, the transposition can occur during token embedding, and the transposed matrix can be processed until the end. For example, flipping the matrix back to its normal form can be done in the final Softmax layer used for classification.

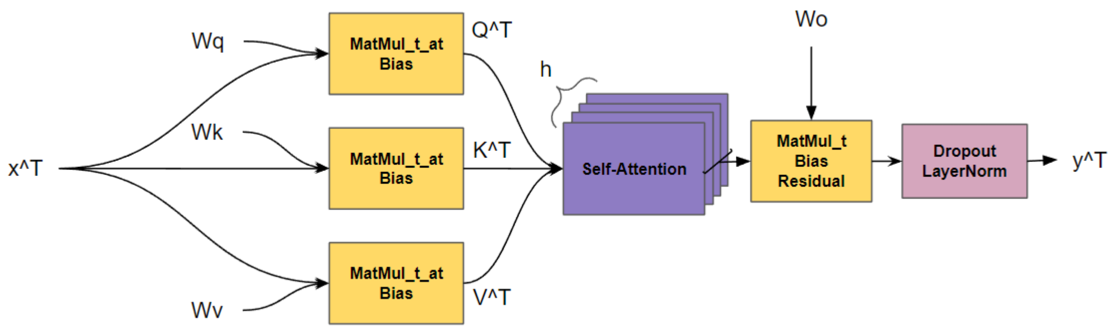


Figure 4.3:

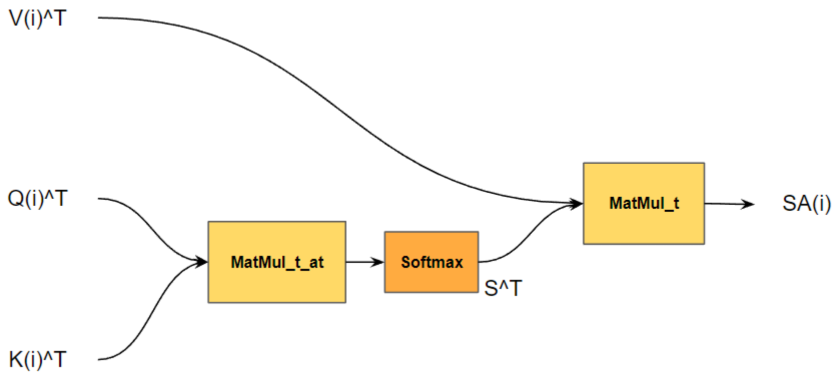


Figure 4.4:

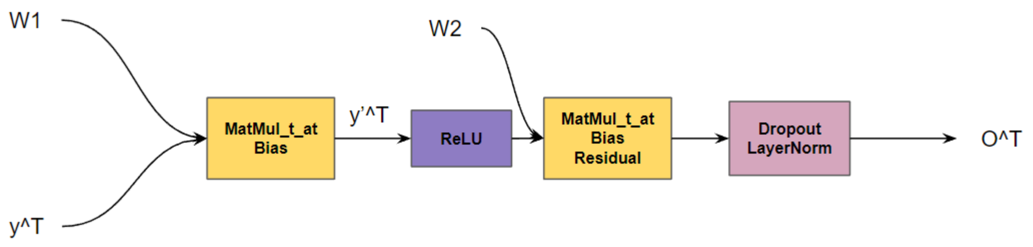


Figure 4.5:

Figure **??** shows the modifications made to Multihead Attention to handle the transposed matrix. We use *MatMul_t_at* to compute all Q/K/V matrices, where *_t* means the result is stored transposed and *_at* indicates matrix operand A is transposed (i.e., input x^T in the figure). Self-attention requires no further transposition as all inputs are already transposed. *MatMul_t_at* (since Q is transposed here) is used to compute $Q * K^T$ and the transposed result can be efficiently processed by Softmax. To get the correct result, we need to multiply S and V. However, both matrices are transposed, so we need to use the property of MatMul, where the transpose of a product is the product of transposed inputs with the operands swapped. $(SA = S \times V,\ \ SA^T = V^T \times S^T)$ And because the new algorithm always transposes the result, we get the self-attention output of each head untransposed. Back to Multihead Attention, the matrix will be transposed again using *MatMul_t*. And in Feed Forward, the matrix is always processed in the transposed form.

In brief, the transposition of the matrix does not affect ReLU or Dropout, while it brings substantial improvement for LayerNorm and Softmax, and MatMul can also benefit from it.

4.2 Hardware

On the hardware side, in order to add support for new instructions, we added some new hardware modules and also made adjustments to the existing modules. In this section we will delve into the details of the hardware implementation of each individual instruction.

4.2.1 Decode & Issue

In the decoding stage, the new instructions are decoded in the dispatcher along with other vector instructions based on the different fields such as "opcode", "func3". The dispatcher records the required operations and vector registers for each instruction and sends the information to the next level. It also handles all CSR reads and writes. The *vsetbl* instruction introduces a new CSR, the broadcast length *bl*. When the dispatcher recognizes the *vsetbl* instruction, it immediately updates the new broadcast length contained in the instruction to *bl*. The data width of each vector register in Ara can range from 8-bit to 64-bit and is set separately for each vector register. The *vfbmacc* instruction only specifies the first vector register, *vd*, but it requires $REUSE\_SIZE$ number of registers. So we use a for loop to iterate through all 32 registers, and if the index is between *vd* and$vd + REUSE_SIZE$, it will be set as well. $REUSE\_SIZE$ is calculated by dividing the vector length by the number of lanes. This also applies to the data hazard tracking before issuing the instruction. The main sequencer adds all used vector registers to the write-after-write (WAW) and write-after-read (WAR) hazard lists.

4.2.2 Broadcast Vector Load

For general vector load instructions, addrgen generates the memory address and sends it to the AXI interface. The returned data is assigned by vldu to different lanes, which then write it to their VRF. However, broadcast vectors are not written in VRF but instead in a unique broadcast buffer (*bc_buffer*). A demultiplexer is added to distinguish whether to send the request to lanes or the broadcast buffer.

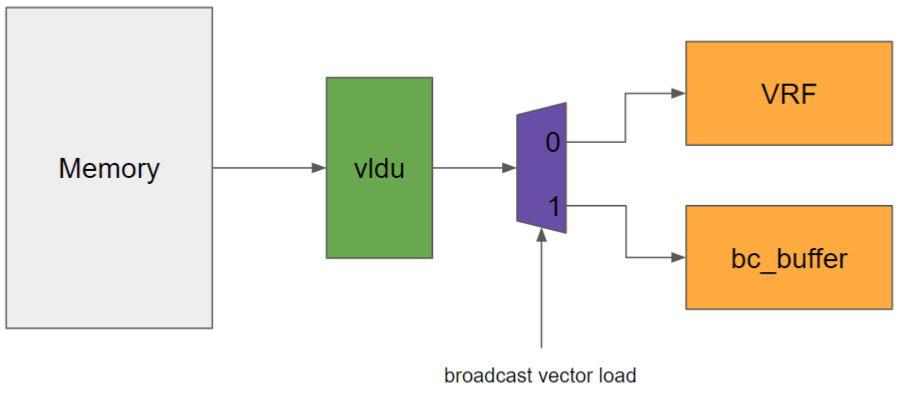


Figure 4.6:

Broadcast Buffer

The *bc_buffer* can be regarded as a special vector register that serves the broadcast vectors. It accepts vector data from *vldu* and stores it in the internal FIFOs, and sends the stored data to the first lane if required. Unlike other vector registers, its read/write mechanism is very simple. *bc_buffer* consists of two 16*64-bit FIFOs, with each can store up to 32 32-bit data. While one FIFO is being read by the processing unit, the other FIFO can be written by *vldu* to store the next broadcast vector. This setup ensures a seamless data flow to the processing unit, as when the current FIFO is used up, *bc_buffer* can immediately provide the next vector by switching to another FIFO.

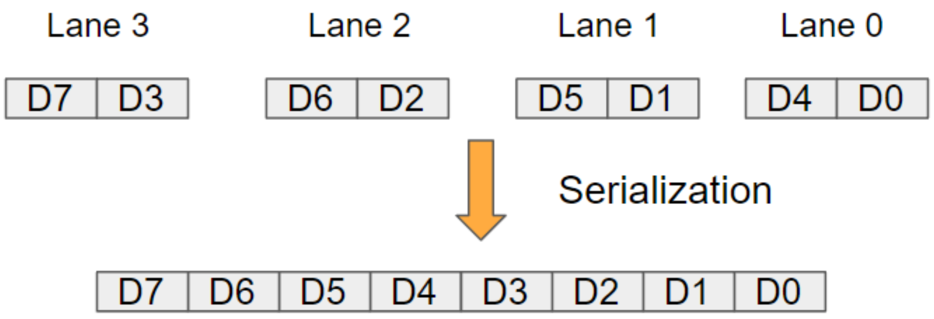


Figure 4.7:

The data bandwidth is 64-bit, but there is a total of *nr_lanes* of signals, i.e. *bc_buffer* receives *nr_lanes * 64-bit* data, and the data are shuffled. In order to process this data, the first step is to serialize all the signals into one signal. This process is depicted in Figure A, where all individual signals are merged into one signal, with the order of the elements arranged from left to right. To determine the number of valid elements in this serialized signal, a "popcount unit" is used to count the valid byte in the byte-enable signal. The FIFO in *bc_buffer* is designed to be read multiple times. Upon receiving the last data from *vldu*, the FIFO switches to the read-only mode. In this mode, the FIFO maintains a state of being full and never empty. When it outputs the last data to the processing unit, it automatically resets the read address to the first data. The first lane plays a crucial role in informing the FIFO that the current vector is finished, allowing the FIFO to clear all data and prepare for incoming new data.

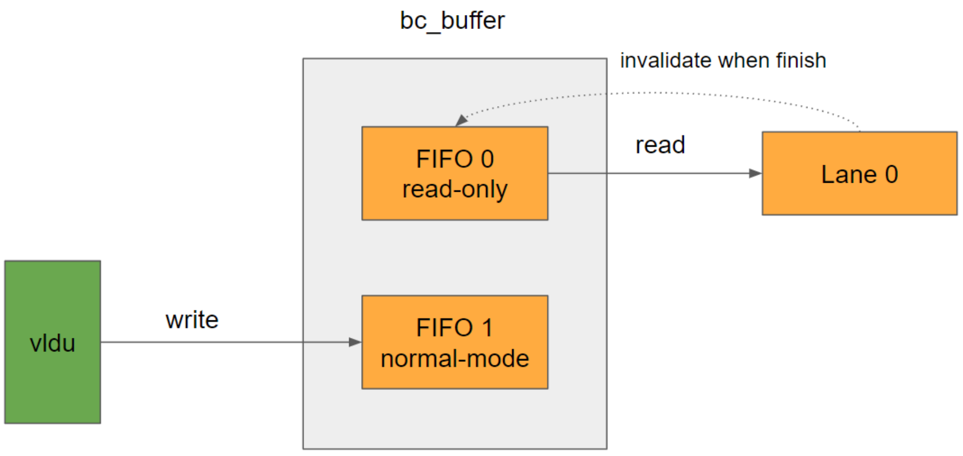


Figure 4.8:

4.2.3 Broadcast MAC

Broadcast Data Path

The broadcast vector contained in the broadcast buffer will be used by all lanes. But sending data to every lane simultaneously can be challenging in the back-end design, particularly with an increasing number of lanes. To make the design more scalable, a pipeline-like data path is used as shown in Figure A. The broadcast buffer only sends data to the first lane, which then buffers it for one clock cycle before forwarding it to the next lane. Since the broadcast signal is buffered in each lane, there is no negative impact on the timing.

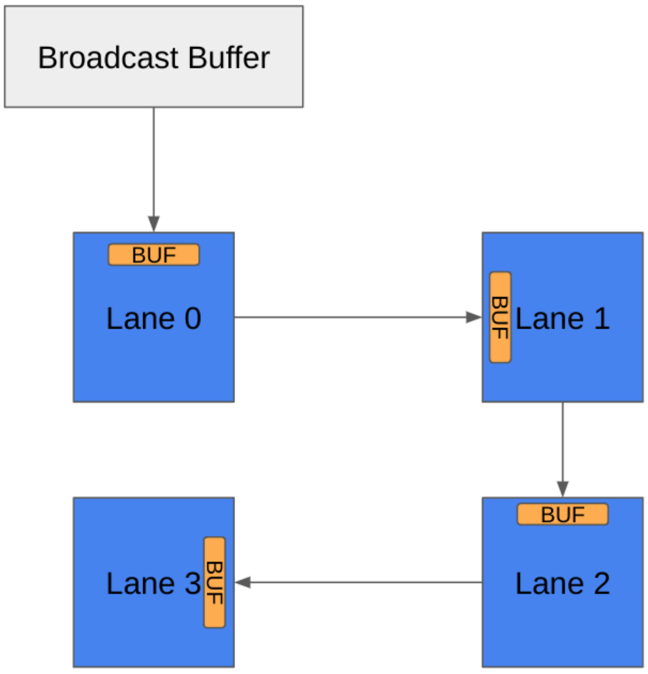


Figure 4.9:

The buffer in each lane is not a simple flip-flop but a two-depth FIFO with a handshake interface, called *bc_operand_queue*, as shown in Figure A. When the current lane receives data from the previous lane, the data will not be immediately sent to the FPU for processing but stored in the FIFO (if FIFO is not full). This is because the delay from the previous lane to the current lane combined with the excessive delay contributed by the FPU itself can have significant performance degradation in the timing. Inserting a FIFO in front of the FPU separates the delay between lanes and the delay from the FPU, reducing its impact on timing.

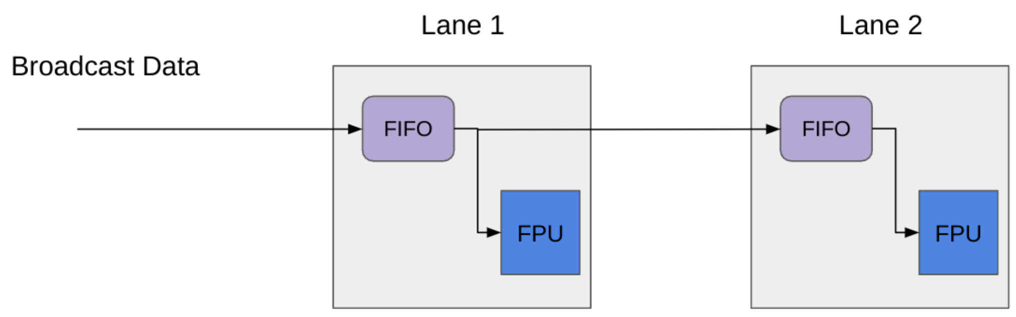


Figure 4.10:

Upon arrival, two scenarios must be considered. The first scenario occurs when the FPU in the current lane is unprepared to receive new data. In this case, transmission will be temporarily halted and the data will be persisted in the FIFO, regardless of whether the next lane is ready or not. The second scenario occurs when the FPU is ready for processing, but the FIFO in the next lane is full and unable to receive new data. The FPU will still process the data, but it will mark it as *data_used*. If *data_used* remains high for the next few cycles, the FPU will be forced to stall. Only when the next lane accepts the current data will *data_used* be reset to low, allowing the current lane to process the next data.

FPU Control

In the FPU's control module, *vmfpu*, a new FSM state is designed to manage the new MAC instruction's operand reading and result processing. The new MAC instruction requires three operands: the a-vector, the accumulated result, and the broadcast b-vector. The first two operands are retrieved from the VRF by the *operand_requester*, while the b-vector is obtained from the *bc_operand_queue* in each lane. The a-vector has a length of *REUSE_SIZE* and each element in the a-vector must be multiplied by the entire b-vector. The result is then stored in a separate vector register after being added to the accumulated result.

The entire computation process can be explained by two for loops, with each having its own corresponding counter. The inner loop multiplies an element taken from the a-vector by the b-vector, and *bc_issue_cnt* counts the number of processed elements from the b-vector ($length = bl$). The outer loop, using *issue_cnt*, counts the remaining unprocessed elements in the a-vector. After each iteration, the index of the destination vector register will be incremented by one to store the next result. When *issue_cnt* reaches zero, the current instruction is complete, and the first lane informs *bc_buffer* that the current b-vector has been fully used.

Operand Request

The operand requester of accumulated results (*vd*) has a distinct behavior compared to other requesters. Unlike the others, the accumulated results come from different vector registers, each with the length of the broadcast length. Similar to the FPU control, the requester is equipped with two counters: one for counting the number of vector registers (*REUSE_SIZE*) and one for counting the elements in one vector register (*bl*). Moreover, we also modified the stall mechanism of the requester to eliminate unnecessary stalls.

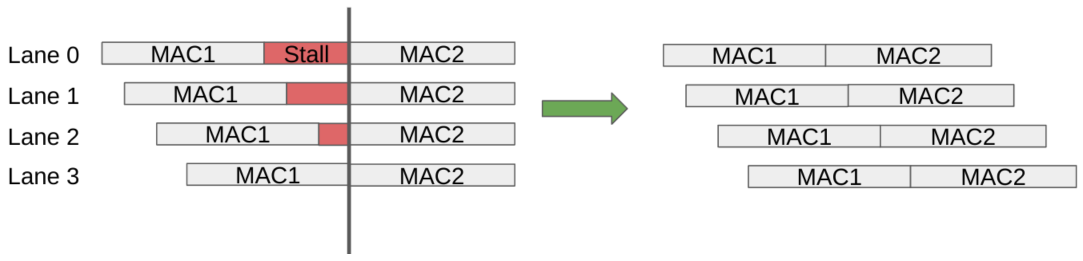


Figure 4.11:

In the original design, the operand requester is stalled if the following conditions are satisfied: 1) The instruction currently being executed by the requester has a data hazard, and 2) The dependent instruction is not writing new data to the VRF. Given the sequential processing of all elements, starting from the first element, when the previous dependent instruction has written a result, the latter instruction is then allowed to require the updated data. Otherwise, the latter instruction must wait for the previous instruction. However, this mechanism is very inefficient for the new MAC instruction, as depicted in Figure A. Consider a scenario where two MAC instructions target the same destination vector register. At the time the first lane completes MAC1, the operand requester of MAC2 becomes stalled as it relies on MAC1, and MAC1 has no more results to write. Although MAC1 is finished in the first lane, it is still active in other lanes, and Ara only considers an instruction to have ended once it has finished across all lanes. As a result, the first lane must wait for the other lanes to complete. And as the number of lanes increases, the stall time increases proportionally. To address this issue, we added an exception to the stall mechanism, enabling each lane to proceed to the next instruction without awaiting the other lanes.

4.2.4 Result Store

The distribution of the broadcast results into vector registers exhibits two properties. Firstly, the results are stored in multiple vector registers, the quantity of which is equivalent to the *REUSE_SIZE*. Secondly, within each lane, the elements in a single vector register are arranged in order, rather than shuffled. Figure A illustrates the distribution of the individual elements of the resulting matrix among different lanes and vector registers. The example depicts a scenario with 4 lanes and *REUSE_SIZE* equal to 2, with a broadcast length (*bl*) of 8. Vector register *vd0* holds the upper half of the matrix, and distinct rows of length bl are stored in different lanes.

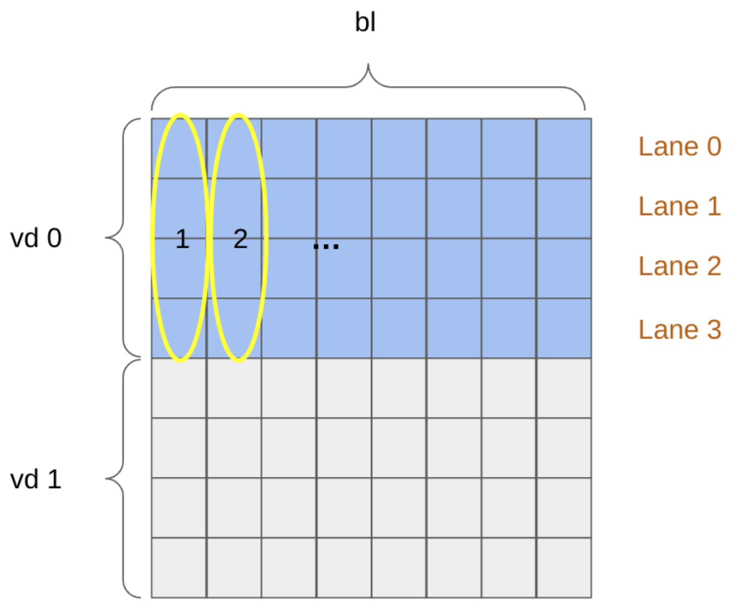


Figure 4.12:

When vstu reads data from vd0, it reads one data from each lane simultaneously to facilitate parallelization. i.e., a single column of the matrix whose length is equal to the number of lanes. To store the elements of this column, a strided store is used, meaning that the address for each element is obtained by adding a stride size to the previous address. However, when storing the first element of the next column, the address is different from the previous rule, it is simply the initial address incremented by one. This leads to two problems. Firstly, the standard strided store is incapable of handling this change, as it can only add the same stride size to the previous address. Secondly, since the addresses are not continuous, only one element can be stored at a time, causing a significant waste of AXI bandwidth. The solution is to transpose the result matrix and implement a different strided store instruction. Once the matrix is transposed, rows are stored instead of columns, and the addresses of elements in a row are contiguous, so the AXI can be used much more efficiently. With the transposed matrix, the new strided store increments the address by the number of columns after each row is stored.

4.3 Physical Design

In the physical design, we mainly optimized the floorplan in the existing backend flow, more specifically, the position of the lane. As shown in the figure, the original floorplan places each lane in order from bottom to top and left to right. Since *lane 1* and *lane 2* are at two opposite corners in the original floorplan, the broadcast path connecting them is not only very long, but also needs to traverse many modules. This path brings a significant delay and is the most critical path in the design. Furthermore, the distance between *lane 0* and *bc_buffer* is also too long, making the delay unacceptable. To reduce the path delay and improve the timing, a new floorplan is proposed. The first lane is placed in the top-left corner, making it closer to *bc_buffer*, and all other lanes are arranged in a ring, minimizing the path length between them.

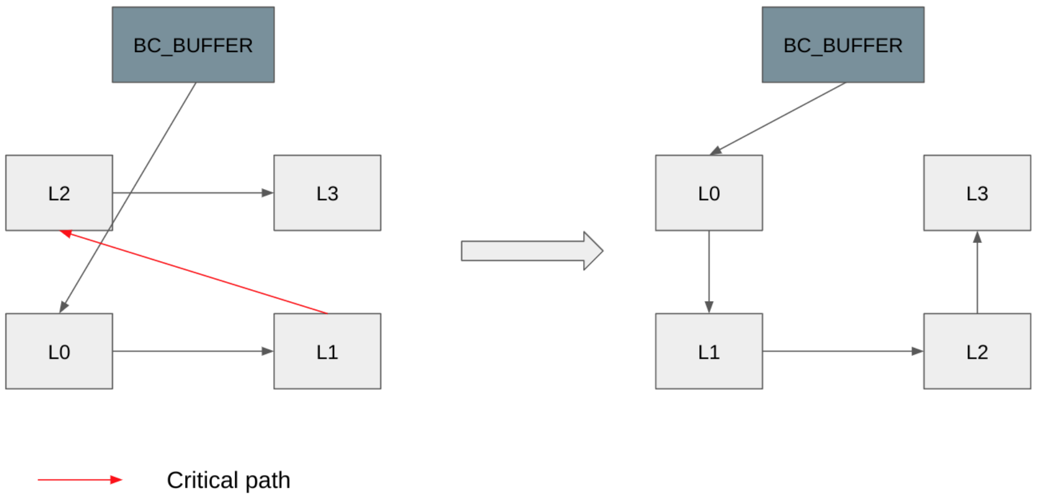


Figure 4.13:

Chapter 5

Results and Discussion

5.1 Performance

We conducted the same performance measurements on the optimized version. The overall execution time distribution is roughly the same, with a heightened MatMul contribution of 98.65%. Figure A illustrates the performance enhancement of each kernel. Since Dropout and ReLU are not affected by the transpose matrix, their performance remains unchanged. LayerNorm and Softmax, on the other hand, benefit from the ability to use unit-stride operations and the burst mode, obtaining performance gains of up to 3.7x and 1.5x, respectively. The refined MatMul also proved to be more efficient at handling smaller matrix multiplications, resulting in a 1.8x increase in the self-attention performance, which consists of two small matrix multiplications and Softmax. However, since the vanilla Ara is already highly efficient in large matrix multiplication, the overall improvement is limited to a mere 5% for 4 lanes and 17% for 16 lanes.

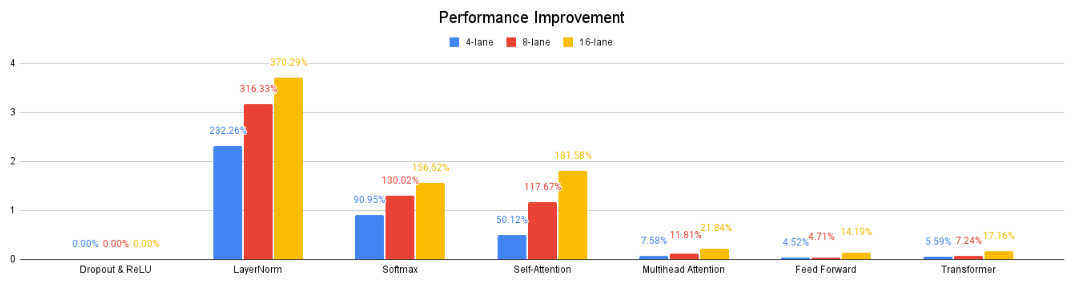


Figure 5.1:

Changing the model size has a similar impact on individual kernels as in the original version. However, when modifying the sequence length, the new version yields greater benefits. As depicted in the figure, when increasing n from 64 to 128, Softmax and LayerNorm both have 8.5% and 7.5% performance improvements, compared to the original's almost unchanged results. Of course, for 4 lanes, further increases in vector length beyond the 128 maximum yield minimal performance impacts.

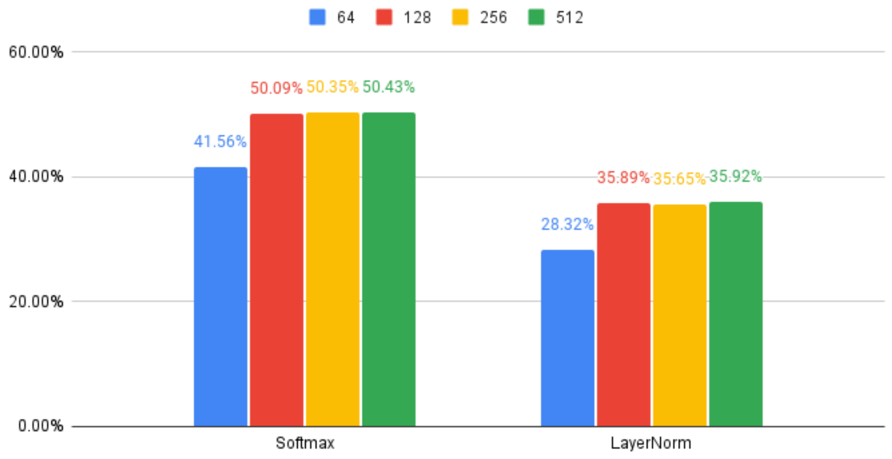


Figure 5.2:

To further understand the benefits of the new MatMul, we performed measurements on matrix multiplications of varying sizes, ranging from $16 \times 16 \times 16$ to $128 \times 128 \times 128$, and tested matrices with different shapes by doubling one of the dimensions ($32 \times 16 \times 16$). Currently, the broadcast path does not allow for interruptions, meaning the current lane cannot halt transmission when the next lane does not have data. So the matrix multiplication has minimum size requirements, with 4 lanes supporting sizes from 16 upwards, 8 lanes supporting 32, and 16 lanes supporting 64.

The utilization improvements are shown in Figure A. The new Ara significantly enhances small matrix multiplication. For example, for 16*16*16 MatMul, the hardware utilization of the new Ara is twice that of the original one. Additionally, the new Ara exhibits reduced sensitivity to the number of lanes, as compared to the original, which experiences a significant drop in utilization with increased lanes, yet the new Ara maintains a high utilization. Hence, the new Ara is more scalable. However, this advantage decreases with increasing matrix size, since the vanilla can use a longer vector to keep the utilization high.

In single-core systems, combining heads in Multihead Attention enhances hardware utilization by extending the vector length. Conversely, for multi-core systems, further parallelization of computation can be achieved by assigning each core to perform the calculation for a different head.

In this scenario, the size of the second matrix operand is reduced from 768x768 to 768x64, causing the application vector length for the vanilla Ara to decrease from 768 to 64. The vanilla Ara performs poorly in this matrix multiplication, especially at 16 lanes, with a hardware utilization rate of only 17%. In contrast, the improved Ara boasts a substantial utilization rate of over 95% even at 16 lanes, a 5.6x improvement over the vanilla. Based on these findings, we calculated the estimated overall execution time for the transformer when the Q/K/V is calculated separately. The performance results for varying lane numbers are displayed in Figure A and show a significant improvement, with the 16-lane improved Ara taking half the time of the vanilla.

5.2 Power

Since Ara has not undergone tape-out yet, We test the power by running the transformer benchmark through gate-level simulation and using the generated waveform file as input for the switching activity in a power analysis tool. However, gate-level simulation takes significantly more time than functional simulation and is not feasible for even the smallest kernel. So we adopted an estimation method by using a small portion of the power from matrix multiplication to represent the overall power consumption. This approach is based on the fact that matrix multiplication accounts for over 95% of the execution time and that large matrix multiplications are composed of MAC instructions of the same vector length, allowing the power of several MAC instructions to be a representative sample of the overall power.

The test results reveal that both the vanilla and improved versions of Ara consume 202 mW and 198 mW respectively. While the improved version consumes more in terms of leakage power, its internal power is lower. The power distribution among individual modules is similar in both versions. 78% of the total power is attributed to the vector processor, while the scalar processor accounts for only 15%. The *bc_buffer*, a newly added module, accounts for only 0.6% of the total power consumption in the optimized Ara. Within the vector processor, modules such as the dispatcher consume a very small percentage, and 90% of the power is consumed by the four lanes. In vanilla, the FPU accounts for a maximum of 57.54% of the lane's power, while in the improved version it is higher, at 69.7%. This increase is because the vanilla Ara requires two register requests from the VRF when running the matmul algorithm, while the improved Ara only requires one, if we ignore the very short a-vector. As a result, the VRF in the vanilla consumes more power and thus the FPU's share is smaller.

We also tested the power of small matrix multiplications with dimensions of 16x16x16 and 32x32x32, and estimated the power for 64x64x64 and 128x128x128 using the same method. When the matrix is very small, the vanilla version consumes less power than the optimized version due to its low performance and low hardware utilization. However, in terms of the performance-to-power ratio, meaning the performance per unit of power, the optimized version has a significant advantage with a maximum improvement of 77%.

5.3 Backend

The vanilla and optimized Ara were implemented using GlobalFoundries 22FDX FD-SOI technology with a clock frequency of 1GHz. The backend process supports both flat and non-flat modes, however, the flat mode was used due to its better results compared to the non-flat mode, which implements a lane separately and integrates the lane as a hardcore IP to save time. Figure A displays the chip layout of the optimized Ara. The green area in the upper half represents the scalar processor, the middle contains various modules of Ara, and the red area is the newly added *bc_buffer* located between vlsu and the first lane.

The optimized Ara has an increased cell area utilization of 82%, a 3.4% increase compared to vanilla, due to the addition of the *bc_buffer* module. The *bc_buffer* module accounts for 2.9% of the total area. Both the optimized and vanilla versions have no negative slack in the typical-case scenario, allowing them to run at 1GHz. In the worst-case scenario, the optimized version has a WNS of -53.43ps ($f_{max} = 949MHz$), while the vanilla version has a WNS of -34ps ($f_{max} = 967MHz$).

Chapter 6

Conclusion and Future Work

In conclusion, this project aimed to vectorize the transformer model and execute it on the vector processor Ara. Through performance analysis, two optimization points were proposed: transposing the matrix to avoid strided memory operations for LayerNorm and Softmax, and introducing a new MatMul algorithm and corresponding hardware to speed up small matrix multiplications and also the individual head calculations of Q/K/V matrices in Multihead Attention. The optimization improves the performance of LayerNorm and Softmax by several times, and also enhances the hardware utilization for small matrix multiplications. When calculating each head individually, the optimized version reduces the overall execution time of the transformer by up to 50%. Additionally, after physically implementing the optimized Ara and comparing it with the vanilla, we found that the new Ara consumes less power during matrix multiplications while maintaining the same overall performance in the backend.

Going forward, there is significant potential for optimization in this project, and several other aspects of the transformer model warrant further exploration. Our future work includes:

1. Support for arbitrary matrices. Currently, the new Ara and algorithm are unable to handle arbitrary matrix sizes. If the length of a-vector is not a multiple of the number of lanes, some lanes may have fewer data. This can cause chaos in the broadcast path as these under-filled lanes are unable to stop receiving broadcast data. A potential solution is to have these lanes acknowledging data, but not utilizing it.

2. Reduce the area overhead of *bc_buffer*. The *bc_buffer* includes two large FIFOs, leading to a 3% increase in the area utilization of the 4-lane Ara and congested chip routing. It is important to evaluate the impact on performance if only one FIFO is used.

3. Support for storage of non-transposed matrices. The current design only supports storing transposed matrices. To accommodate for non-transposed results, a new

store instruction must be introduced. Due to limited coding space, a new opcode to hold all new instructions is needed.

4. Further improve utilization. The new algorithm currently faces two limitations in terms of utilization: it cannot perform computational instructions during store instructions and the *REUSE_SIZE* is fixed to 4 in experiments. To overcome the first limitation, we can load the data for the next iteration into the VRF earlier, allowing the FPU to continue functioning without interruption. To address the second limitation, we can consider increasing the *REUSE_SIZE*, though this will require more vector registers.

5. Directly apply bias and residual connection. Since MatMul produces results that span multiple rows and columns, applying bias or residual values directly to the results before they are written back requires more effort. Two possible solutions are to store the values in memory in a format that is compatible with the MatMul algorithm or to load them with a specialized load instruction. Additionally, there are several pipeline opportunities within the transformer model that can be leveraged, such as applying Dropout and ReLU operations after MatMul, just like the bias term.

6. Investigate novel MatMul algorithms. While our new algorithm modifies the way data is processed, it does not alter the essence of the original algorithm. There are multiple faster MatMul algorithms that have the potential to reduce computational complexity, such as the Strassen's algorithm [27] which leverages the cost advantage of addition over multiplication, or the Alpha Tensor [28] which uses AI and reinforcement learning to increase computation speed by 20%. It would be exciting to explore the possibility of adapting these algorithms for use in vector processors.

7. Explore the sparsity. DNNs often exhibit high sparsity, meaning many values are zero. Skipping computation of these zero values can significantly improve performance and reduce power consumption. Current vector processors use mask vectors to ignore certain data, but the data is still processed, leading to wasted power and time. Eyeriss v2 [29] implements a specialized sparsity format that only stores non-zero weights in memory, and avoids calculation for zero activations. A similar sparsity mechanism could be applied to Ara, enhancing its efficiency as a transformer accelerator.

8. Use a more proper transformer model. For an edge processor like Ara, the transformer model used in the experiments is very unsuitable in terms of size. A more reasonable approach is to use a model similar to KWT (usually containing only 1-3 heads) [30]. As the new hardware and software are very efficient on small matrices, we expect more performance improvements when running small transformers.

List of Acronyms

ASIC	application-specific integrated circuit
DFT	design for testability
DZ	Microelectronics Design Center at ETH Zurich
FPGA	field-programmable gate array
IC	integrated circuit
IIS	Integrated Systems Laboratory
PDF	Portable Document Format
SNR	signal-to-noise ratio
SPSE	Situation-Problem-Solution-Evaluation
SVG	scalable vector graphics
VCS	version control system
WYSIWYG	“what you see is what you get”

List of Figures

List of Tables

Bibliography

[1] OpenAI, “Chatgpt website,” https://openai.com/blog/chatgpt/.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[5] IBM, “What are neural networks?” https://www.ibm.com/uk-en/cloud/learn/neural-networks.

[6] A. Afshine and A. Shervine, “Recurrent neural networks cheatsheet,” https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks.

[7] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” *arXiv preprint arXiv:2010.11929*, 2020.

[8] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, “Swin transformer: Hierarchical vision transformer using shifted windows,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 10 012–10 022.

[9] S. Mehta and M. Rastegari, “Mobilevit: light-weight, general-purpose, and mobile-friendly vision transformer,” *arXiv preprint arXiv:2110.02178*, 2021.

[10] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linformer: Self-attention with linear complexity,” *arXiv preprint arXiv:2006.04768*, 2020.

[11] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang *et al.*, “Big bird: Transformers for longer sequences,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 283–17 297, 2020.

[12] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[13] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5156–5165.

[14] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 10–14.

[15] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 2, pp. 1–341, 2020.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[17] H.-T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 01, pp. 37–46, 1982.

[18] M. Shaaban, "Eecc756: Multiple processor systems," http://meseec.ce.rit.edu/cmpe655-fall2015/.

[19] S. Lu, M. Wang, S. Liang, J. Lin, and Z. Wang, "Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer," in *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*. IEEE, 2020, pp. 84–89.

[20] X. Yang, B. Yan, H. Li, and Y. Chen, "Retransformer: Reram-based processing-in-memory architecture for transformer acceleration," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[21] F. Tu, Z. Wu, Y. Wang, L. Liang, L. Liu, Y. Ding, L. Liu, S. Wei, Y. Xie, and S. Yin, "A 28nm 15.59 μj/token full-digital bitline-transpose cim-based sparse transformer accelerator with pipeline/parallel reconfigurable modes," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 466–468.

[22] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[23] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, and L. Benini, "A "new ara" for vector computing: An open source highly efficient risc-v v 1.0 vector processor design," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2022, pp. 43–51.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[25] A. M. Rush, "The annotated transformer," in *Proceedings of workshop for NLP open source software (NLP-OSS)*, 2018, pp. 52–60.

[26] "Spike risc-v isa simulator," https://github.com/riscv-software-src/riscv-isa-sim.

[27] V. Strassen *et al.*, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[28] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R Ruiz, J. Schrittwieser, G. Swirszcz *et al.*, "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, no. 7930, pp. 47–53, 2022.

[29] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[30] A. Berg, M. O'Connor, and M. T. Cruz, "Keyword transformer: A self-attention model for keyword spotting," *arXiv preprint arXiv:2104.00769*, 2021.