A MASE-Based LLM.int8 Hardware Quantization and Acceleration Architecture on FPGA *

*Project 3 : LLM.int Hardware Integration for MASE

1st Moteng Ma
Dept. Electrical and Electronic Engineering
Imperial College London
London, United Kingdom
mm1823@ic.ac.uk

1st Zixian Jin
Dept. Electrical and Electronic Engineering
name of organization
London, United Kingdom
cj323@ic.ac.uk

***Abstract*—This paper presents and implements hardware quantization and acceleration architectures for LLM inference on FPGA based on the paper LLM.int8 [1]. Most quantization and acceleration design is based on GPUs, but customized and optimized FPGAs can save more power and reduce latency. The main challenge of deploying quantization and acceleration design on FPGA is that the RTL design is more complex than configuring CUDA on GPU. This work is trying to deploy the LLM.int() algorithm on FPGAs and design a block-wise quantization method to adapt to the interface of FPGAs. Through timing simulation and synthesis, our quantization and acceleration architecture shows a higher resource efficiency and time efficiency.**

***Index Terms*—LLM, mixed-precision quantization, streaming accelerator design, int8.**

I. Introduction

In recent years, the improvement of CPU computing power has encountered the bottleneck due to power consumption constraints and dark silicon effect, while deploying large language models (LLMs) has put intensive computational and memory requirements for hardware. As a result, GPUs and FPGAs that can perform parallel computing play the main roles on forward inference and backward propagation computations of LLMs.

Due to the mature ecosystem on GPUs, previous research [1], [2] mainly focused on utilizing GPUs to accelerate the computation of LLMs. However, the high performance computing capabilities of GPUs come with high energy consumption, which faces the difficulties as the parameters of LLMs increase. Compared to GPUs, FPGAs can provide higher energy efficiency, especially when tasks can be highly customized through hardware acceleration. Another advantages of FPGAs is that FPGAs can provide optimized solutions and accelerate the computation process for applications requiring low-latency processing.

The main challenge on applying FPGA acceleration to LLMs is that the computation and memory resources are limited. For matrix multiplication, splitting the large matrix into small matrices for parallel input is a classic acceleration method. However, this optimization strategy faces memory bound, which means the memory resource utilization is higher than computational resource utilization so that memory resources often face shortages on FPGAs. To solve this limitation and balance the memory and computing resource utilization, quantization is used to cut the memory needed for LLMs.

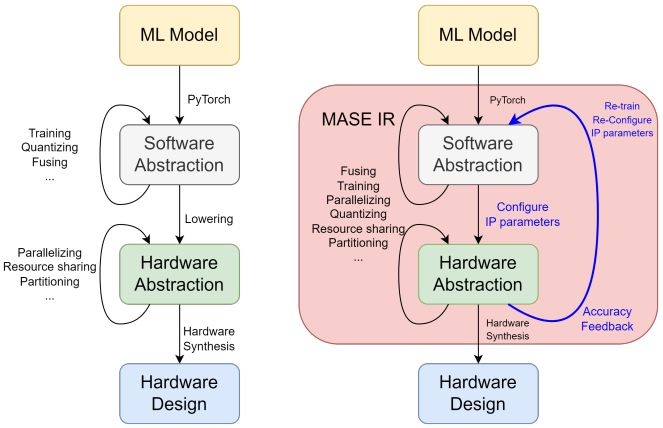

Fig. 1. Schematics of MASE toolflow.

Although FPGAs offer high flexibility, allowing developers to redeploy LLMs and optimize performance and energy efficiency for specific applications. Redeploying LLMs on FPGAs still faces the difficulty in dynamic updates and design complexity. Compared to platforms like GPUs, FPGAs may not be as flexible and quick when the model or algorithm needs updates. Although some FPGAs support field-programmable features, the updating process is still more complex than software-level updates.

The complexity of custom-designed hardware accelerators on FPGAs means their update frequency struggles to keep pace with the rapid advancements of machine learning (ML) models. As a result, Machine-Learning Accelerator System Exploration Tools (MASE) [3], [4] is developed to co-design

Moteng Ma, CID: 02485216.
Zixian Jin, CID: 02469417.

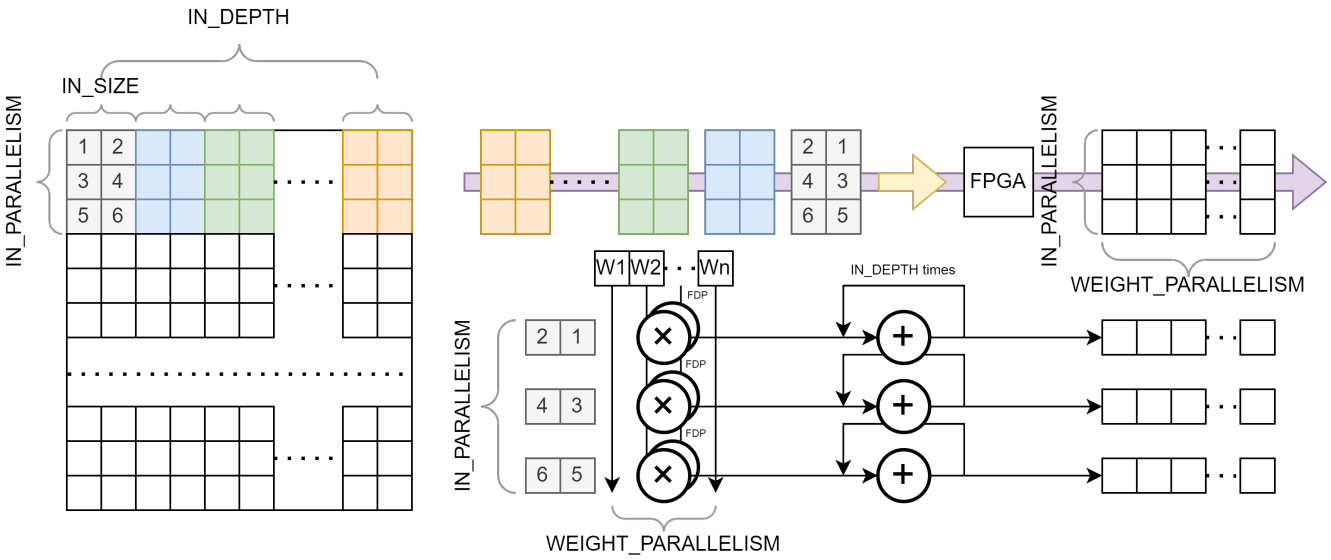

Fig. 2. Schematics of matrix multiplication for LLM.

the software models and hardware accelerators. This paper presents the MASE-based quantization method and hardware architecture for matrix multiplication of LLM on FPGAs.

In this paper, the Section II summarizes the related work. In Section III introduces and develops the quantization and optimization methods for LLM matrix multiplication. Finally, Section IV demonstrates and evaluates our work.

II. Related Work

A. Quantization

The quantization for LLMs not only accelerates the computation but also makes a major contribution to resource utilization. As we show in Section I, since the GPUs are designed for highly parallel computational tasks, They allow for the simultaneous processing the large scale of data and make the LLM computation highly efficient. As a result, most previous related works are based on GPUs to develop the general quantization techniques.

Before the advent of ChatGPT and other LLMs, quantization is mainly developed for Convolutional Neural Networks (CNNs). Early research [5] concentrated on quantization-aware training (QAT), which attempted to improve quality by retraining. When it comes to LLM quantization, this quantization method causes the resource utilization to be high. By contrast, post-training quantization (PTQ) can keep the quality without retraining, which is suitable for large scale parameters computation.

Since only 8-bit datatype is supported by GPUs [1] , the LLM.int8 quantization method is submitted. Absmax quantization is a classic quantization that records the maximum value of the matrix and scales the activations into 8-bit range by [-127, 127]. Although the range of int8 is [-128, 127], the reason for discarding -128 is to maintain symmetry, which avoids the bitwidth to be changed in absolute operation. Compared to absmax quantization, ZeroPoint quantization can be better adapt to the actual distribution of data, potentially leading to less information loss and thus maintaining higher model accuracy. ZeroPoint scaling is a quantization technique that utilizes linear scaling and ZeroPoint offset to distribute floating-point data more evenly across the int8 range.

For W8A8 quantization, Tim Dettmers et al. [1] focused on the vector-wise quantization and considered to utilize mixed precision quantization to deal with outliers so that the quantization method can maintain the quality. Zhewei Yao et al. [6] studied group-wise quantization for weights and token-wise quantization for activations, but the layer-by-layer knowledge distillation caused the computation time to increase. Guangxuan Xiao et al. [7] presented the SmoothQuant, which can share the difficulty of quantization between weights and activations. Compared to the work of LLM.int8, This method can maintain accuracy while significantly speeding up inference. These quantization methods show that rough quantization can still maintain accuracy through algorithms. In contrast, careful quantization strategies often cause larger latency without great improvement in accuracy.

For lower precision quantization, especially for int3/int4, the research [8]–[10] focused on designing algorithms and architectures to maintain accuracy and accelerating inference computation. Due to the high clock frequency of the GPUs, these algorithms and architectures can be implemented quickly, which do not lead to a significant increase in latency. However, when developers design the hardware and implement these complex algorithms and architectures at the register-transfer level (RTL), timing will become challenging and dependent data will waste time waiting the pre-process of other data, which causes the number of cycles to increase. Since

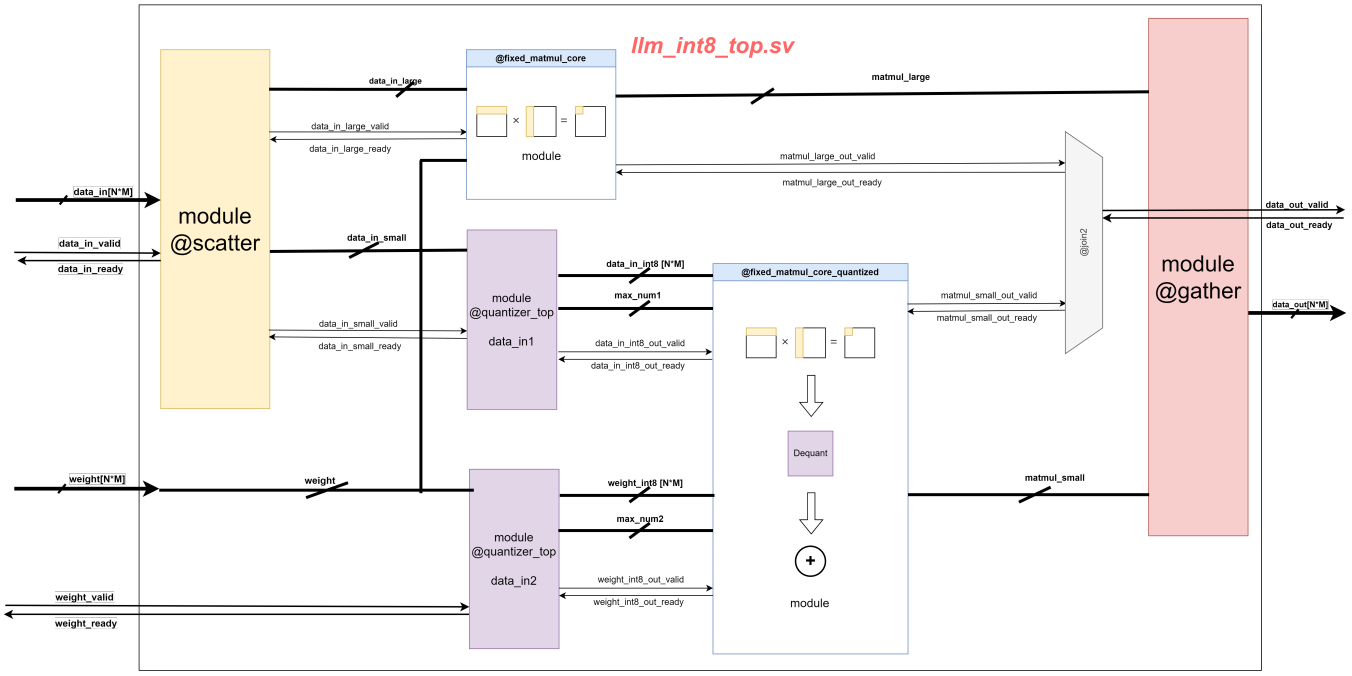

Fig. 3. Schematics of top-level architecture of LLM.int8 matmul on FPGA.

the clock frequency of FPGAs is slower than GPUs, these increased cycles cause the latency to significantly increase. As a result, most previous related works are based on GPUs.

When it comes to deploying LLMs on FPGAs, it is necessary to begin with the simple algorithms. The RTL design involves careful consideration of timing issues, control signal propagation and other hardware-specific constraints. FPGAs can be fine-tuned to create highly efficient pipelines and parallel data paths that match the exact timing and processing requirements of an application.

B. Matrix Multiplication of LLMs

Compared to general matrix multiplication, the large scale matrices in LLMs need to be split into small scale matrices and sent into FPGAs by data package and double buffer. According to the Fig. 2, the data in the small scale matrices is fed into the fixed point dot production modules and accumulators to get the result of the large matrix multiplication.

Since pruning and quantization will cause the matrices to become sparse, the acceleration on sparse matrix multiplication is important for reducing latency. Wenjie Li et al. proposed an architecture called matrix-vector multiplications (MVMs) of LLMs to achieve better area efficiency and energy efficiency.

III. Method

In Section II, we have shown the difficulties when deploying the LLMs on FPGAs and designing the architecture on RTL. This section, we present the methods that are applied to solve these challenges.

A. Architecture Overview

1) Streaming Dataflow: The Fig. 4 shows the mixed-precision decomposition process in this paper. According to the data figures in LLMs [1], outlier features appear systematically in large models in a way that they either occur in a column or not at all. Another data features in LLMs is that the outlier features only appear in activations rather than weight, which means the weight is easier to quantize than activations [7]. This motivates a design of streaming LLM matrix column-wise, such that all elements in a column can be counted as outliers once there is only one outlier spotted in that column. This distribution pattern facilitates hardware design and resource utilization, since the hardware scatters matrix in a column-wise instead of element-wise manor. However, according to the parameters of ChatGPT and other LLMs, the size of an LLM matrix is typically from thousands to thousands. Feeding a whole column to computation unit on FPGAs is infeasible as it has limited I/O ports and on-chip memory blocks. In this paper, the LLM matrix to be computed is divided into several sub-matrices in both the row and column dimension, as shown in Fig. 2. Each sub-matrix has the size $M * N = IN\_PARALLELIM * IN\_SIZE$ where M denotes the row size and N denotes the column size. By tuning M and N, the sub-matrix can be smaller enough to be input to FPGAs for computing. The parameterized port definitions are more conducive to the automatic generation and tuning of the hardware.

2) Pipeline: One of the biggest advantages of dataflow-based accelerator is that the computation is deeply pipelined, leading a high throughput. In this paper, all sequential-logic

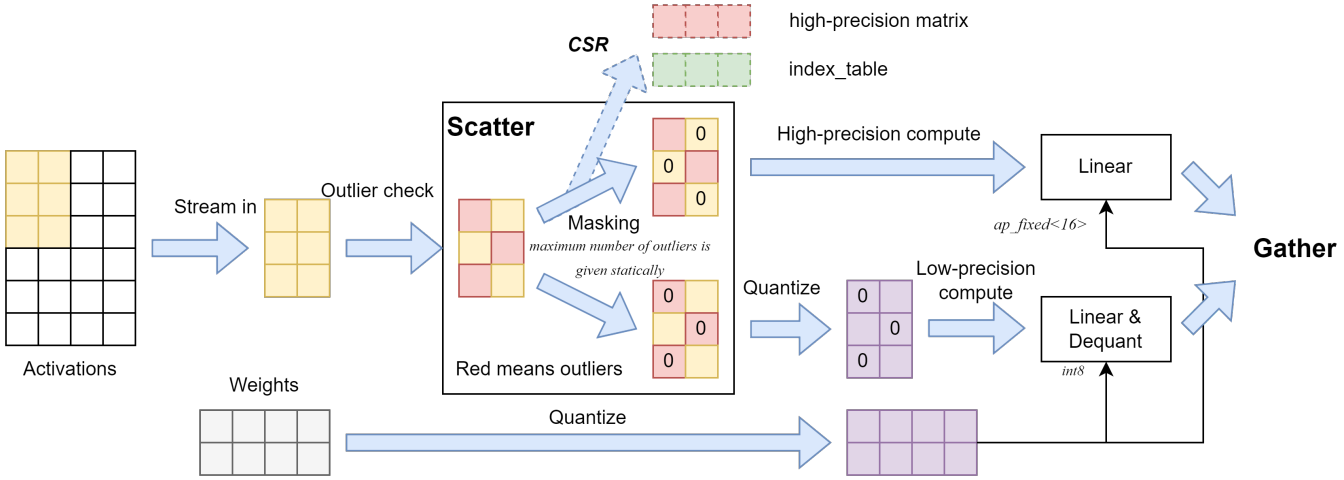

Fig. 4. Schematics of top-level data stream.

modules are pipelined and use handshake protocols to transfer data between up-stream and down-stream stages.

In the following section, we introduce the micro-architecture of modules.

B. Scatter

The core idea of the previous work on GPUs [1] is to decompose an LLM matrix into matrices of different magnitudes hence compute them with customized precision. To implement this logic in FPGAs, a scatter module is designed and placed before the matrix multiplication module. This scatter module detects large-magnitude elements in the input matrix X_{f16} and output them to a matrix $X_{HP,f16}$ for high-precision computation. The rest of the entries are allocated to another matrix $X_{LP,f16}$ for low-precision operation.

There are three design parameters for this module, which are critical to performance of the entire LLM.int accelerator. The following paragraphs show a rough analysis of the parameters. Quantitative results of their impact will be discussed in Section IV.

1) Threshold for magnitude of outliers.: The first deisgn parameter is $LARGE\_NUM\_THRES$, which defines the threshold for identifying a value as large-magnitude one. In the LLM.int8 algorithm [**?**] the input matrix is of precision FP16, and the scattered low-precision matrix is of Int8 . All entries in the low precision matrix are quantize by a absmax quantization method:

$$X_{i8} = \left\lfloor \frac{127 \cdot X_{f16}}{\max\left(\left|X_{f16_{i,j}}\right|\right)} \right\rceil = \left\lfloor \frac{127}{\|X_{f16}\|_{\infty}} \cdot X_{f16} \right\rceil = \lfloor s_{X_{f16}} \cdot X_{f16} \rceil \tag{1}$$

where $\lfloor\rceil$ indicates rounding to nearest integer. The tensor is divided by its absolute maximum magnitude and then multiplied by 127 to ensure that all elements fit in the range [-127, 127] of Int8 precision.

Therefore, the maximum possible values $max(X_{LP,f16})$ to during absmax quantization is always less than the parameter MAX_NUM_THRES. If the parameter is set too large, the scale factor $s_{x_{f16}}$ would become much smaller than 1, causing significant loss when the tensor X_f16 is scaled and rounded. However, if the parameter is instead too small, most of elements in the input matrix is counted as outliers, meaning more FP16 computing units are consumed. For the case of our project, the default value for MAX_NUM_THRES is set as 127, which is equal to the maximum range of Int8 representation. Scaling factor $s_{x_{f16}}$ is no less than 1 by such setting, so rounding loss is expected to be less than 1.

2) Size of the scattered matrices.: There are basically two ways of storing the HP and LP matrices $X_{HP,f16}$ and $X_{LP,f16}$ coming out of the scatter module.

The first way is to make the two scattered matrices completely dense and only store effective elements. As shown in Fig. 4, only large-magnitude elements in X_{f16} will be stored in $X_{HP,f16}$, while the remaining normal elements will not find a place in $X_{HP,f16}$. Under such setting, the size of $X_{HP,f16}$ is much smaller than X_{f16} if there are only few large-magnitude values in the LLM matrix, as [1] suggests. This saves computation resources for the HP matmul module. However, the size of $X_{HP,f16}$ would be highly dynamic at run-time for different input matrix, since the position of large-magnitude values in X_{f16} is irregular between rows and columns. To handle this irregularity, advanced format such as Compressed Sparse Row (CSR) must be used to represent $X_{HP,f16}$. The existing linear or matmul components in MASE does not support such compressed data representation. This makes hardware design much more difficult, and also introduces hardware resource overhead for supporting such compressed storage format.

Alternatively, the design that our project implements is a naïve but efficient way. $X_{HP,f16}$ and $X_{LP,f16}$ are produced by applying different masks on the input X_{f16}. The 1's in the HP mask means the corresponding entry in X_{f16} has a large

magnitude and should be placed in $X_{HP,f16}$. After masking, large-magnitude entries in X_{f16} are preserved at the same position in $X_{HP,f16}$, while the rest entries in $X_{HP,f16}$ are all zeros. This sparse matrix are still treated as dense matrices for matrix multiplication. The dimension of the scattered matrices, therefore, are identical to the input matrix. This leads to a simple hardware design as the size of I/O ports and computation units is statically determined at compile-time for scatter and matmul. Compared to the first method, this design is not efficient in terms of hardware resource, as it does not exploit sparsity of the HP matrix. However, this disadvantage can be reduced or even hided, since we can tune the number of outliers to be placed in the $X_{HP,f16}$ such that it becomes dense. This strategy is related to another design parameter MAX_LARGE_NUM, which will be discussed below.

3) Limited number of identified outliers.: The LLM.int8() algorithm [1] in the paper detects all large-magnitude values in the input X_{f16} and allocates them to the HP matrix $X_{HP,fp16}$. In hardware level this is implemented as $N*M$ parallel large-number checkers, where $N*M$ denotes the input matrix size. This design has two limitations. First, checking all entries in parallel consumes a huge amount of hardware resources. Second, it cannot control how many large-magnitude numbers preserve high precision, so it does not help in comprehensively evaluating the LLM.int8() mixed-precision quantization strategy. Therefore, a parameter MAX_LARGE_NUM is used in our design. There are two cases for different input matrix:

- Actual outlier number $<= MAX\_LARGE\_NUM$: All outliers go into the HP matrix, which is identical to the paper's algorithm [1]. The LP matrix only contains small-magnitude values so applying Int8 quantization on it does not bring too much loss.
- Actual outlier number $> MAX\_LARGE\_NUM$: Only part of outliers go into the HP matrix. The LP matrix contains several large-magnitude numbers. They interfere the Absmax Quantization, as the scaling factor $s_{x_{f16}} = 127/max(|X_{f16}|)$ is greatly decreased and causes rounding loss.

If setting $MAX\_LARGE\_NUM = 0$, then all large-magnitude values go into the LP matrix, meaning all elements in the input matrix X_{f16} is quantized to Int8 precision. This simulates the conventional 8-bit quantization method. By tuning MAX_LARGE_NUM we expect to observe the performance difference of LLM.int8() mixed-quant with conventional quant strategies in terms of accuracy.

It also worth noting that the limited outlier checking also benefits the design of highly efficient sparse HP matrix computation. The issue of irregular HP matrix size that we discussed previously can be solved here since the HP matrix has a statically known size MAX_LARGE_NUM. Exploiting the matrix sparsity is then feasible using compressed storage format. We leave this architecture implementation as future work.

C. High-precision processing unit

High-precision matrix is fed into the fixed_matmul_core module, which is computed by fp16 matrix multiplication. The only thing worth noting is how to achieve synchronization of high-precision matrix multiplication and low-precision matrix multiplication through control signals. This is a join2 module in Gather to achieve synchronous input.

D. Low-precision processing unit

The Low precision processing unit consists of quantizer modules and a int8 matrix multiplication integrated with de-quantization.

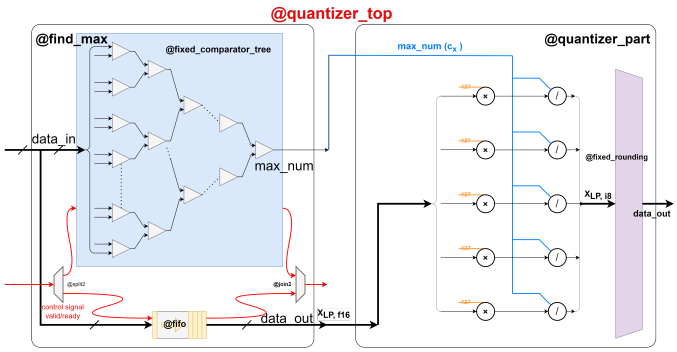

Fig. 5. Schematics of quantizer module.

1) quantizer: According to the Fig. 5, the quantizer_top module consists of the find_max module and the quantizer_part module. The find_max module is used to find the absolute maximum data. To accelerate this process, the search process is conducted in parallel so that the comparator-tree module is designed. This module can speed up the process about finding the maximum data, but it also causes the time latency, which delayed by $log[IN\_SIZE * IN\_PARALLELISM]$ cycles. As a result, FIFO is required in order to wait for the comparator-tree to locate the maximum data.

Compared to the vector-wise quantization [1], the block-wise quantization is more suitable for FPGA design. Due to the limited resources of FPGAs and limited bandwidth for transmitting data between PL and PS on FPGAs, it is impossible to compute a column in parallel simultaneously. Another reason is that the input of the entire column vector ignores the correlation between rows, which means each data in one row has a different scaling factor. The block-wise quantization ensures that the length and width of the input matrix can be adjusted. This provides a foundation for self-adaptively defining port parameters through software models on MASE.

To keep the quality, both quantizers and dequantizers must be sequentially passed through multipliers, bitwidth extensions, left shifts, and dividers. Since data need to be quantized, first rounding, dequantized and second rounding. Bitwidth extensions and left shifts can ensure rounding towards the nearest number instead of directlly cutting off the excess decimal bits.

First multiplying and then dividing is also to ensure that the bit width is expanded by multiplication and only needs to be rounded once. Otherwise, first passing through the divider causes that rounding within all data in the small representation range, which leads to a decrease in accuracy.

This is also the reason why we just transmit the maximum data from quantizer to dequantizer rather than scale factor. Since the scale factor is a decimal number, it needs more bits to be represented. For example, if the maximum is just 1, the scale factor is $\frac{1}{127}$, which has 56 decimal bits in its binary representation.

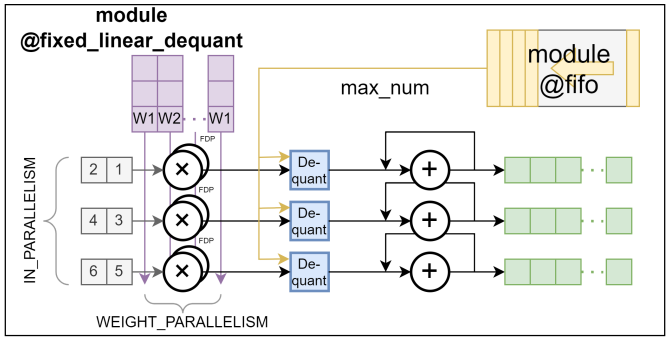

Fig. 6. Schematics of matrix multiplication integrated with de-quantization.

2) matrix multiplication integrated with de-quantization: Since each sub-matrix generates a maximum number, each block has its own maximum number to perform de-quantization before accumulating the results of dot product in one row of activations. As a result, the de-quantization for each block is performed before accumulation. Contrary to quantization, de-quantization multiplies the maximum data and divide by the maximum representation range of int8, which is 127.

E. Gather

During scatter, we just set 0 for the data that reaches the threshold without changing the dimension of matrix. As a result, the gather module in hardware is just matrix addition of the output matrices of the HP and LP processing units.

IV. Evaluation

We evaluate the design in three dimensions. First, we evaluate the functionality of the top module as well as sub-modules. This is done by Cocotb testbench. Specifically, we evaluate the computation accuracy of our design by comparing the LLM.int-quantized output data with theoretical output data computed with full-precision on Cocotb. Second, the latency of the LLM.int component is evaluated also by Cocotb. Third, the circuit area, or equivalently hardware resource utilization is analysed via Xilinx Vivado synthesis and implementation.

A. Scatter and Gather Functionality

According to Fig. 4 and Fig. 7, the activation data is fed into the scatter module which is used to select the data if it is needed to be quantized. The simulation results show that the activation matrix is split into 2 sub-matrices: the high-precision matrix and the low-precision matrix. The high-precision matrix is computed by the fp16 matrix multiplication and the low-precision matrix is fed into quantizer and then sent to int8 matrix multiplication.

The simulation results also show that the maximum number of outliers is given statically. The MAX_LARGE_NUMBER is set by 3, so the number of data in the high-precision matrix cannot be greater than 3. This causes the dimensions of the high-precision matrix to be fixed, thus facilitating using the acceleration methods.

Fig. 7. The waveform of the scatter module.

B. Performance Analysis

There are two streams of dataflow between the scather and gather operation: the HP processing unit and the LP one. The handshake signals of the two streams need to be synchronised before joining to the gather component. Therefore, the total throughput of the LLM.int component is limited by the worst throughput among the two processing units.

The HP unit is basically a FP16 matmul computation. The module "quantized_matmut" is the top-level module of the low-precision processing unit. It takes the LP matrix $X_{LP,f16}$ and W_{f16} as input, and performs 8-bit quantization, matrix multiplication and de-quantization that we discuss in III-D. Since both the HP and LP processing units are fully pipelined, the throughput is only limited by the matrix accumulator, which sums up dot-product results of vectors in the same row in the un-partitioned LLM matrix. We set the accumulation parameter $IN\_DEPTH = 3$ and obtain the timing diagram of the LP processing unit and of the entire LLM.int component as shown in 8 and 9, respectively. The two input matrix are streamed in every clock cycle, and the output data is valid every three cycles. Therefore, the throughput of this module is equal to the theoretical throughput IN_DEPTH.

C. Computation Accuracy Analysis

As we discuss in III-B, there are two design parameters for the scatter components that affect the accuracy of the LLM.int unit: MAX_LARGE_NUM, which defines the maximum number of outliers placed in HP matrix

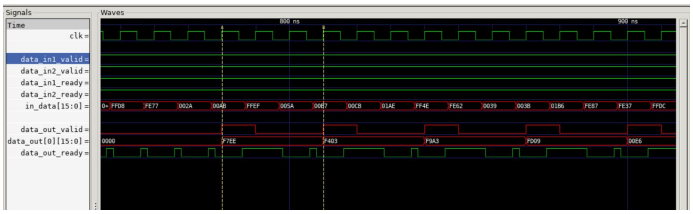

Fig. 8. The timing diagram of the low-precision processing units.

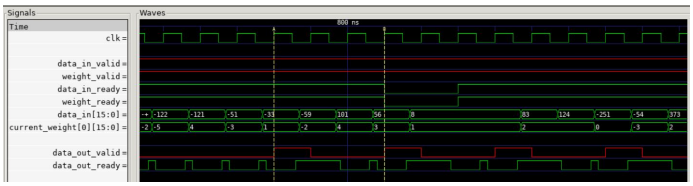

Fig. 9. The timing diagram of the LLM.int8 module.

and $LARGE\_NUM\_THRES$, which defines the maximum magnitude to be counted as outliers.

11 and 12 present the quantitative evaluation of the two design parameters. As decreasing MAX_LARGE_NUM or increasing $LARGE\_NUM\_THRES$, more large-magnitude values are pushed to the LP matrix. Therefore, the quantization constant c_x and c_w increase, which are the absolute maximum value of the two matrices. This can greatly decrease the matrix computation accuracy. This side effect is most significant when extremely large values exists in the LP matrix, as shown in the right-most data points of the two figures. For normal settings of MAX_LARGE_NUM and $LARGE\_NUM\_THRES$, the absolute maximum error is below 50, which means an maximum relative error of roughly 2%.

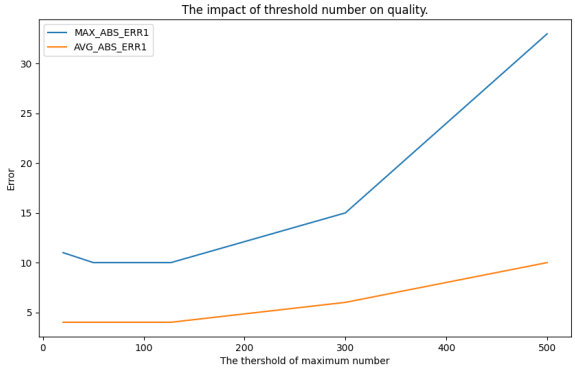

Fig. 10. The impact of the threshold value on absolute error.

D. Hardware Utilization Analysis

After the simulation, we verify and test the IP core by running synthesis. Fig. 13 shows the relationship between the number of parameters in LLMs and the DSP utilization on FPGAs. LUT_2 represents the LUT utilization of fixed_matmul_core. LUT_n% represents the LUT utilization

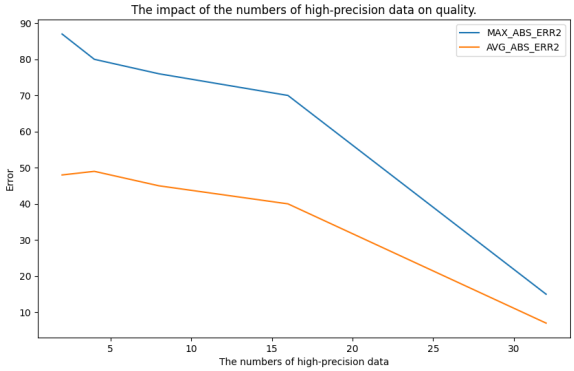

Fig. 11. The impact of the number of high-precision data on absolute error.

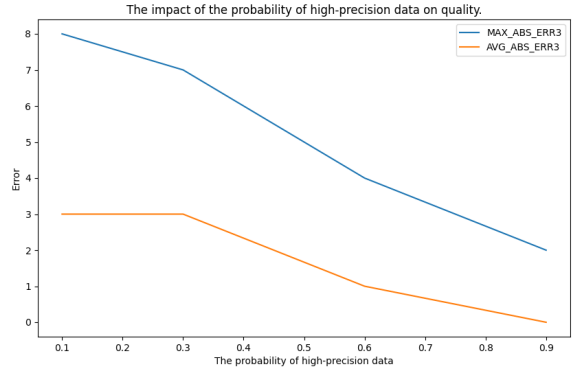

Fig. 12. The impact of the probability of high-precision data on absolute error.

when the high-precision data probability is n%. For FPGAs, $int8 * int8$ is computed by LUT, while $fp16 * fp16$ is computed by DSP.

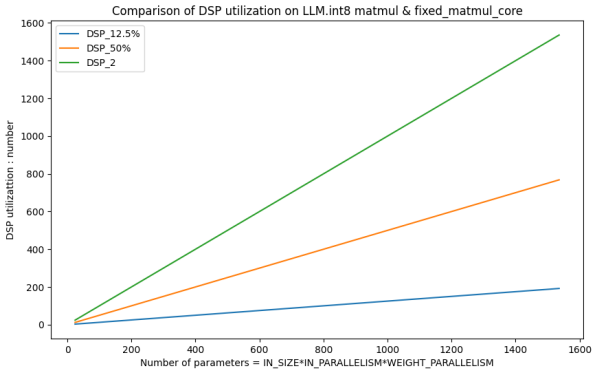

Fig. 13. Comparison of DSP utilization.

Utilizing LUTs to replace DSP to complete multiplication can reduce the latency and power consumption. It is noticeable that the number of high precision data is equal to the number of DSP used.

Fig. 14 shows the relationship between the size of block-wise quantization matrix and the LUT, FF utilization on FPGAs. Compared to the general matrix multiplication, LLM.int8 matrix multiplication has a higher resource efficiency and time efficiency.

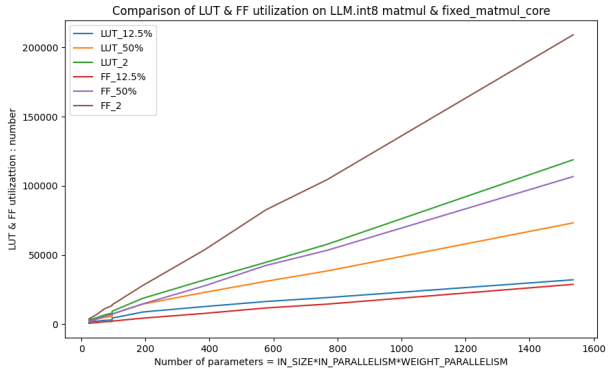

Fig. 14. Comparison of LUT and FF utilization.

Resource utilization shows a linear increase trend, as the number of parameters in block-wise matrix increases. This is because the data is computed in parallel. The storage of the data array is completely partitioned and the computation process is fully unrolled. The Fig. 15 shows the relationship between resource utilization and the activation depth of LLMs. The depth of activation has little impact on resource utilization since its meaning is related to the depth of pipeline.

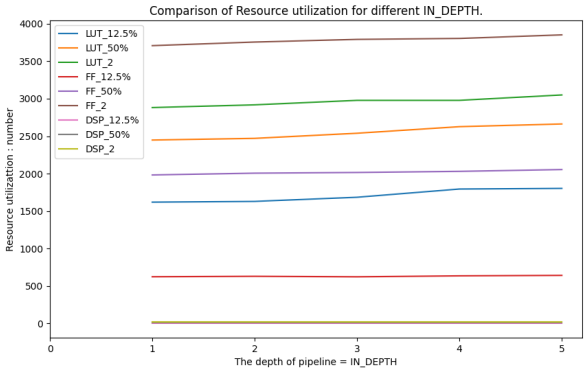

Fig. 15. Comparison of Resource utilization for different IN_DEPTH.

E. Limitations

1) Other quantization methods: Since this work is based on RTL design for FPGAs, we choose the simple absmax quantization methods. Zero-point quantization and other quantization methods on GPUs can maintain the quality while reducing the resource efficiency and latency. However, complex logic and architecture may cause timing issues on FPGAs, which causes the difficulties when designing hardware at RTL. In the future, we will try another quantization methods and solve the issues encountered when designing these methods on FPGAs.

2) Implementation: The time efficiency and dynamic power consumption can not be verified by running on FPGAs, but they are estimated theoretically. The main reason for this difficulty is that the AXI module in MASE library does not have detailed configuration instructions. When running implementation, the I/O interface cannot be set correctly. In the future, we will create the interface module. We will verify and test the actual results on FPGAs.

3) Sparse Matrix: Since both quantization and pruning will cause the matrix to become sparse, applying sparse matrix multiplication is a classic acceleration methods on GPU accelerator for LLMs. Fortunately, another project is ongoing about the sparse matrix multiplication for LLMs. In the future, we will try to utilize the methods on that project to solve this limitation.

Conclusion

This paper present a LLM.int8 block-wise quantization matrix multiplication for LLMs on FPGAs. The previous work mainly focused on quantization and accelerating GPUs computing LLMs through CUDA. This is a new attempt to design quantization and hardware architecture for LLMs at RTL. The experimental results show that this quantization and acceleration method has a higher resource efficiency. We estimate this method also has a higher time efficiency theoretically.

References

[1] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

[2] Wenjie Li, Aokun Hu, Ningyi Xu, and Guanghui He. Quantization and hardware architecture co-design for matrix-vector multiplications of large language models. *IEEE Transactions on Circuits and Systems I: Regular Papers*, pages 1–14, 2024.

[3] Jianyi Cheng, Cheng Zhang, Zhewen Yu, Alex Montgomerie-Corcoran, Can Xiao, Christos-Savvas Bouganis, and Yiren Zhao. Fast prototyping next-generation accelerators for new ml models using mase: Ml accelerator system exploration, 2023.

[4] Cheng Zhang, Jianyi Cheng, Zhewen Yu, and Yiren Zhao. Mase: An efficient representation for software-defined ml hardware system exploration.

[5] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients, 2018.

[6] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers, 2022.

[7] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[8] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[9] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. In *MLSys*, 2024.

[10] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization, 2024.