OPTIMUS: OPTIMIZED MATRIX MULTIPLICATION STRUCTURE FOR TRANSFORMER NEURAL NETWORK ACCELERATOR

Junki Park [1] Hyunsung Yoon [1] Daehyun Ahn [1] Jungwook Choi [2] Jae-Joon Kim [1]

Abstract

We present a high-performance Transformer neural network inference accelerator named OPTIMUS. OPTIMUS has several features for performance enhancement such as the redundant computation skipping method to accelerate the decoding process and the Set-Associative RCSC (SA-RCSC) sparse matrix format to maintain high utilization even when large number of MACs are used in hardware. OPTIMUS also has a flexible hardware architecture to support diverse matrix multiplications and it keeps all the intermediate computation values fully local and completely eliminate the DRAM access to achieve exceptionally fast single batch inference. It also reduces the data transfer overhead by carefully matching the data compute and load cycles. The simulation using the WMT15 (EN-DE) dataset shows that latency of OPTIMUS is 41.62×, 24.23×, 16.01× smaller than that of Intel(R) i7 6900K CPU, NVIDIA Titan Xp GPU, and the baseline custom hardware, respectively. In addition, the throughput of OPTIMUS is 43.35×, 25.45× and 19.00× higher and the energy efficiency of OPTIMUS is 2393.85×, 1464× and 19.01× better than that of CPU, GPU and the baseline custom hardware, respectively.

1 Introduction

In recent years, neural machine translation based on deep learning has been widely used. Recurrent neural network (RNN), and long short-term memory (LSTM) have been popular choices for machine translation (Sutskever et al., 2014; Cho et al., 2014; Bahdanau et al., 2015). However, RNN/LSTM are known to have some problems; it is hard to parallelize the computation due to sequential characteristics (Wu et al., 2016a) and the accuracy drops when the input sentence is very long (Cho et al., 2014). The attention mechanism improves the accuracy by allowing the decoding process to focus on the input part which is the most relevant to the current decoding step (Bahdanau et al., 2015). In particular, the Transformer neural network which consists of attention mechanisms only is known to have much more parallelism and improved translation quality (Vaswani et al., 2017).

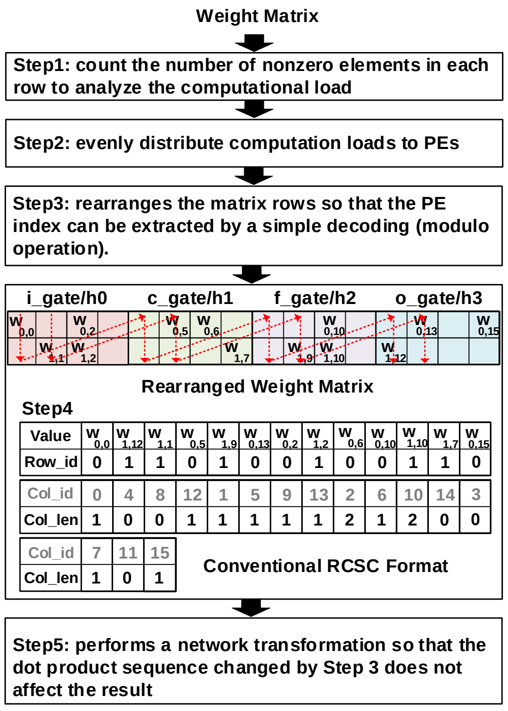
While various inference hardware accelerators for RNN and LSTM have been proposed (Han et al., 2017; Gao et al., 2018; Wang et al., 2018; Park et al., 2018; Park et al., 2019; Cao et al., 2019), there is a lack of research on hardware to accelerate the inference of the Transformer despite having better performance than RNN and LSTM. There are several challenges in designing a transformer inference engine. First, the overhead of DRAM access is large because of the large amount of data. A well-known technique called pruning can be applied to reduce the memory requirement (Han et al., 2015a; 2017). Second, when large number of multiplier and accumulators (MAC) are embedded in the accelerator to increase the parallelism and the performance, MAC utilization is reduced. This problem is exacerbated when the dense weight matrix becomes sparse after pruning. Third, the computation flow of encoding and decoding in the Transformer is very different and the excessive computational overhead in decoding should be addressed. In the encoding process, all the word vectors in a sentence are computed in parallel as a matrix form. However, only one word vector is translated for each decoding iteration. Since all previously decoded word vectors need to be used as an input to the decoder at the next decoding step, the amount of computation increases quadratically during the iterations.

This paper presents a high-performance and flexible hardware architecture, OPTIMUS, for the transformer algorithm inference. The main contributions of the paper can be summarized as follows:

1. We **analyze the computation process of the Transformer network and improve the performance by skipping redundant computations**. It is shown that sequential generation of words in the Transformer decoder is the bottleneck in terms of performance and skipping redundant

[1]Department of Creative IT Engineering, Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea [2]Department of Electronics and Computer Engineering, Hanyang University, Seoul, Republic of Korea. Correspondence to: Jae-Joon Kim <jaejoon@postech.ac.kr>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020. Copyright 2020 by the author(s).


Weight Matrix
Step1: count the number of nonzero elements in each row to analyze the computational load
Step2: evenly distribute computation loads to PEs
Step3: rearranges the matrix rows so that the PE index can be extracted by a simple decoding (modulo operation).
i_gate/h0
c_gate/h1
f_gate/h2
o_gate/h3
$W_{0,0}$ $W_{0,2}$ $W_{0,5}$ $W_{0,6}$ $W_{0,10}$ $W_{0,13}$ $W_{0,15}$
$W_{1,1}$ $W_{1,2}$ $W_{1,7}$ $W_{1,9}$ $W_{1,10}$ $W_{1,12}$
Rearranged Weight Matrix
Step4
Conventional RCSC Format
Step5: performs a network transformation so that the dot product sequence changed by Step 3 does not affect the result

Value	$W_{0,0}$	$W_{1,12}$	$W_{1,1}$	$W_{0,5}$	$W_{1,9}$	$W_{0,13}$	$W_{0,2}$	$W_{1,2}$	$W_{0,6}$	$W_{0,10}$	$W_{1,10}$	$W_{1,7}$	$W_{0,15}$
Row_id	0	1	1	0	1	0	0	1	0	0	1	1	0

Col_id	0	4	8	12	1	5	9	13	2	6	10	14	3
Col_len	1	0	0	1	1	1	1	1	2	1	2	0	0

Col_id	7	11	15
Col_len	1	0	1

Figure 1. The process of generating the conventional RCSC format. It solves the problems of load imbalance and input load miss caused by a sparse matrix.

computations reduces the overhead significantly. We also show that skipping redundant computations is much more effective in custom hardware design than in GPU.

2. We **propose a Set-Associative RCSC (SA-RCSC) format** to enable large-scale MACs to maintain high utilization. The proposed sparse matrix format significantly reduces the input miss rate by allowing multiple PEs to handle a matrix row. As a result, the MAC utilization is improved by $\sim$ 2X compared to the conventional RCSC format case.

3. We **design the OPTIMUS, a custom hardware accelerator for the Transformer neural network** which has a flexibility to support various types of matrix multiplications. While it outperforms generic computing platforms by significant margin in general, OPTIMUS shows a particularly good performance for a single batch inference by keeping all the intermediate computation values fully local and eliminating DRAM access. It also has an optimized control flow to hide the data transfer overhead from the computation.

2 Background and Related Work

2.1 Sparse Neural Machine Translation

Neural machine translation (NMT) is to map a sequence of words in one language to one in another language using a neural network based sequence to sequence model. In general, the sequence to sequence model consists of two parts, encoder and decoder, where encoder extracts the time-varying feature of the input sentence and decoder exploits it to predict a sentence, a word at a time. There have been various approaches for constructing encoder and decoder. LSTM-based layer structures such as Google's Neural Machine Translation (Wu et al., 2016b) have been popular for their superior translation performance, but they suffer restricted parallelism inherent in LSTM computation. Recently, a network based primarily on the attention mechanism, the Transformer (Vaswani et al., 2017), has been introduced to increase parallelism in computation.

The state-of-the-art NMT models are often composed of multiple layers with large weight matrices. Therefore, model compression such as pruning (Han et al., 2016; 2017) is commonly used to alleviate the memory access overhead for loading weights. After weight elements with small importance are pruned to zero, a dense matrix becomes sparse. To eliminate the overhead of fetching unnecessary zero elements, a pruned weight is stored using a sparse matrix format such as a compressed sparse column (CSC) format (Han et al., 2017), which consists of the non-zero values, row indices and column pointers of non-zero elements.

However, two major problems arise when the CSC format is used for the sparse matrix computation in the custom hardware accelerators. First, since the computation load is unevenly assigned to each PE, the overall PE utilization is reduced. Second, since the input vector element is loaded from the input buffer in irregular access pattern, the miss rate of the input is high. If the corresponding element is not loaded from the input buffer due to a miss, the PE is stalled until the corresponding input element is loaded. There have been several studies to solve these load imbalance problem and input load miss problem (Han et al., 2017; Park et al., 2018; Rizakis et al., 2018; Park et al., 2019). Among them, only the rearranged compressed sparse column (RCSC) format proposed in (Park et al., 2019) addresses both issues.

2.2 Rearranged Compressed Sparse Column Format (RCSC)

The RCSC format (Park et al., 2019) utilizes the characteristics of LSTM to improve the hit rate of the input vector in the local buffer as well as balancing the computation loads between PEs. This format was introduced as a sparse matrix format targeted for LSTM, but is applicable to all networks in which an input vector is multiplied to multiple sparse weight matrices.

The RCSC format is generated through a five-step process (Fig. 1) (Park et al., 2019). The first step (Step 1) is to analyze the computation load for each PE by counting the number of nonzero elements in each row. The second step (Step 2) is to assign a PE for each row. The computation load is evenly distributed to each PE in this step. The third step (Step 3) is to sort the matrix rows in circular order based

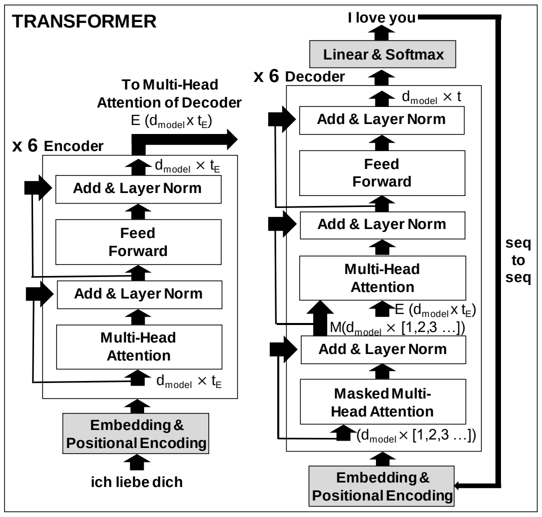


Figure 2. Model architecture of the Transformer.

Table 1. The Computation Type of the Transformer

1. EMBEDDING AND POSITIONAL ENCODING		
EM/PE	$E = Embedding(X) + PE(X)$	
2. MULTI-HEAD ATTENTION		
COM1	$[Q, K, V] = [W^Q, W^K, W^V] \cdot Y$	WEIGHT(sM)
COM2	$P = K^T \cdot Q$	WEIGHT(dM)
COM3	$S = Softmax(P/\sqrt{d_k})$	
COM4	$Z^{0-7} = V \cdot S$	WEIGHT(dM)
COM5	$Z = W^O \cdot Concat(Z^{0-7})$	WEIGHT(sM)
3. RESIDUAL ADDITION AND LAYER NORMALIZATION		
LN	$Z = \gamma Norm(Y + Z) + \beta$	
4. POSITION-WISE FEED FORWARD		
FF1	$Z = ReLU(W^{F1} \cdot Z + b^{F1})$	WEIGHT(sM)
FF2	$Z = W^{F2} \cdot Z + b^{F2}$	WEIGHT(sM)

on the PE index assigned to compute each row. In Step 4, the first columns of weight matrices for the 4 LSTM gates are encoded before the second columns are encoded. This encoding order increases the probability of having non-zero weight values in adjacent columns. In the Transformer architecture, a similar approach can be applied to the multi-head attention case, in which an input is multiplied by multiple weight matrices. The fifth step (Step 5) is to transform the weight matrix so that rearrangement does not affect the outcome. How the RCSC format is applied to the Transformer is described in detail in Section 5.

2.3 Transformer Neural Network

The Transformer is one of the most popular neural machine translation methods thanks to its superior performance and the improved parallelism. Yet there is limited study on its computation patterns to design customized accelerators. In this section, we provide a brief explanation of the computational characteristic of the Transformer, with the key computations summarized in Table 1. (We ask readers to refer Appendix A for more details.)

The Transformer has the form of encoder-decoder (Fig. 2). One sentence composed of t_E words is represented by a $d_{model} \times t_E$ matrix when the embedding and positional encoding are finished. The matrix of these symbol representations is computed over six encoder layers. When the encoding is finished, the output containing the encoding information becomes a key-value pair of the multi-head attention in the decoder layers. While a whole input sentence is processed in parallel in the encoding layers, decoding of an output sentence is done word by word as the decoding of each word requires the previously decoded words as the input. Thus, decoding for an encoded sentence requires repeated computations of all decoder layers. The output from each decoding iteration is the probability of the word following the previous word. This process is repeated until the end of the sentence (EOS) is decoded.

Here we briefly explain the computation patterns in the Transformer. Each encoder layer is composed of two sub-layers: multi-head self attention layer and position-wise fully-connected feed forward layer. Each decoder layer has one more sub-layer: masked multi-head attention. The masking ensures that the prediction of output word depends on the previous output words only. All these layers are followed by the residual connection and layer normalization.

Multi-head attention is the structure to measure the relationship among words in the sentence. This process is divided into five computations (COM1∼5) in Table 1. COM1 is a matrix-matrix multiplication that computes query (Q), key (K), and value (V). COM2 is to compute the score which represents how relevant each word is to other words. COM3 is to scale down the value in order to stabilize gradients during training (Vaswani et al., 2017). COM4 is to multiply the result of COM3 by value (V). COM5 is to concatenate the results (Z^0 - Z^7) of each head and multiply the concatenated results by the weight matrix (W^O) to mix them. In the position-wise feed forward network of each layer, two linear transformations are executed, which the first one involves Rectified Linear Unit (ReLU) activation. Residual addition and layer normalization are inserted after each (masked) multi-head attention and feed forward network.

3 CHALLENGES FOR TRANSFORMER ACCELERATION

3.1 Limited Parallelism in Decoder

In the Transformer, the computation pattern in the encoding stage is vastly different from the decoding stage. In the encoding stage, all the words in an input can be processed in parallel thanks to the attention-based layer structure – there is no dependency via hidden states across the time-steps

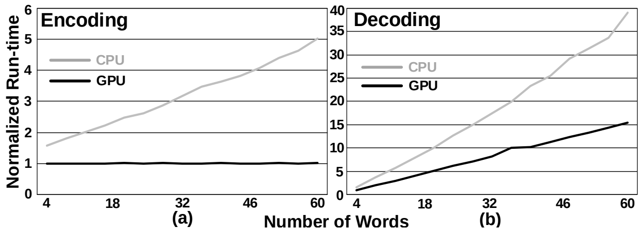


Figure 3. The CPU and GPU processing time for different numbers of words. (a) In encoding process, all words are computed in parallel. (b) In decoding process, word is sequentially decoded one by one.

in encoder. Therefore, one can exploit parallelism in the time-step dimension to accelerate the processing speed. For example, one can stack word vectors into an input matrix and employ matrix-matrix multiplication to reuse weight matrix and perform computation in parallel across the time-step. Since the decoder shares the similar layer structure with the encoder, there is no hidden state dependency in it. However, the decoder still suffers limited parallelism since in the decoding stage the computation for the prediction at one time-step depends on the prediction of all the previous time-steps. Such dependency requires a feedback structure in the computation along the time-step dimension, leading to repetitive load of weight for each time-step and slow speed even with the parallel processing units.

The challenge of the limited parallelism in the decoding stage is demonstrated in Fig. 3, where the processing time for the encoding and decoding stages is compared for CPU (multi-thread) and GPU. In the encoding stage, the processing time increases as the sentence length grows for CPU while it is almost constant for GPU. This implies that the amount of computation needed for more number of words is fully parallelized using GPU once the weight is loaded. Therefore, GPU can achieve high speedup over CPU when encoding long sentences. In contrast, the speedup of GPU over CPU is much lower in the decoding stage. This indicates that the overhead of repetitive load of weight due to the limited parallelism in decoder shows up as the number of words increases and limits the effectiveness of the GPU implementation.

3.2 Low MAC Utilization

In real-time applications, latency is a very important design specification. For example, when the machine translation is applied to the simultaneous interpretation, the translation latency of each sentence (batch size = 1) must be very short. On the other hand, when multiple users perform translations (batch size > 1) via a server at the same time, throughput for multiple batches becomes an important specification. In summary, reducing latency when processing a single batch and increasing throughput when processing multiple batches are one of the key design issues in accelerator design.

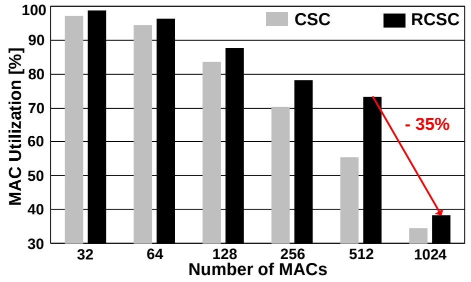


Figure 4. Average MAC utilization for Transformer. The MAC utilization degrades significantly as the number of MAC increases in both CSC and RCSC formats.

In order to improve latency and throughput, accelerators need to have large number of MACs. However, as the number of MAC increases, the load imbalance and input load miss problems caused by the sparse matrix become more serious. Although the RCSC format mitigates these problems somewhat, low MAC utilization still limits the maximum performance in hardware accelerator when many MACs are used (Fig. 4). This paper proposes an extension to the RCSC format to maintain high utilization even when a large number of MACs is used. The detailed explanation will be given in Section 5.

4 Skipping Redundant Decoding Computations

As discussed in Section 3, the computational complexity of decoding layers increases over time-step due to the feedback structure of the network. Note that in the decoding stage the output word in the previous time-step comes in as a new input token to the network, which is stacked into a input matrix. Input word or output word becomes token as expressed as a vector that becomes the input of encoder or decoder after the process of embedding and positional encoding. Fig. 5a shows the detail computation procedure in Masked Multi-Head Attention layer in the decoding stage (cf. Fig. B.1 for Multi-Head Attention layer). Note that the input token at time-step t is being stacked to $Y = [y_1, y_2, ..., y_t]$. This stacking is necessary since the correlation between K and Q is computed over the entire time steps in COM2. Due to the stacking, the computational complexity as well as the amount of data needed for the computation increase linearly as the time-step increases. This results in quadratic increase of the total decoding operations as well as the data elements, as demonstrated in Fig. 6 for various sentence length.

However, if we carefully investigate the computation procedure in the decoding stage, it can be noticed that the unique information added at each time step is constant except for COM2 and COM4, as highlighted in Fig. 5b. More specifically, if we maintain K and V for all the previous time-steps, we can compute COM2 and COM4 without performing re-

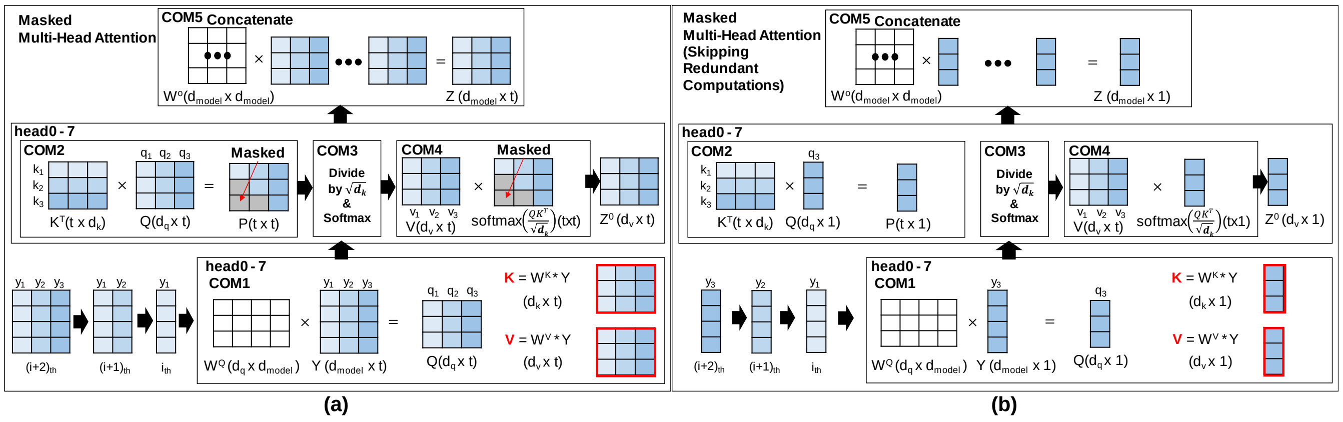


Figure 5. Comparison of the computing flows for the masked multi-head attention between (a) the conventional flow with on-the-fly iterative computations and (b) the proposed flow with redundant computation skipping

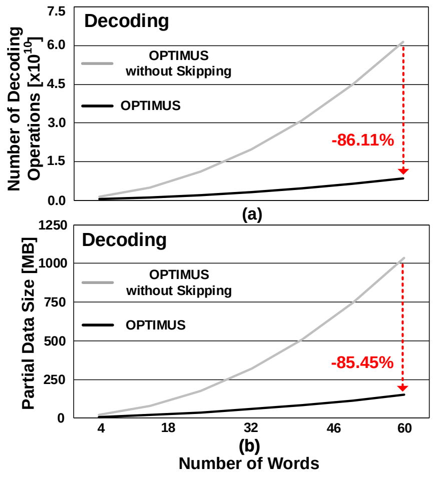


Figure 6. (a) Comparison of the number of decoding operations depending on skipping computation. (b) Comparison of partial data size depending on skipping computation.

dundant computation of re-creating them in COM1. Note that computation in COM1, COM3 and COM5 takes time-step as an independent dimension. Therefore, once K and V of the previous time-step are loaded, token vectors only for the current time-step, i.e., q_t, k_t, v_t, need to be newly computed to produce z_t, which will be used as the new token for the next layer.

This change allows us to *skip redundant decoding computation*, and there are three implications with it. First, since K and V of previous time steps are loaded (rather than computed on the fly), it increases memory load overhead. But its overhead is much smaller compared to loading weights, since the typical size of K and V (e.g., $K[d_k \times t]$) is much smaller than the weight (e.g., $W^K[d_k \times d_{model}]$) where $t < d_{model}(= 512)$. Furthermore, there are savings as we need to keep the input token for the next layer Y just for one time-step. Therefore, the overall increase of memory load overhead is small.

Second, this change opens up the possibility of keeping intermediate activation fully local. As shown in Fig. 5b, the storage needed for intermediate activation is (almost) independent to t (i.e., the size of buffer needed for keeping $Z^0[d_v \times 1]$ is independent to t and P is typically smaller than Z^0). This implies that one can assign a fixed buffer size to keep all the intermediate activation locally and avoid DRAM memory access.

The third implication is that the computation pattern in decoding is changed from Matrix-Matrix to Matrix-Vector multiplication. This change becomes a serious issue for GPU. As demonstrated in Section 7, GPU cannot exploit the benefit of skipping redundant decoding computation as it suffers seriously low utilization for Matrix-Vector computation. Whereas, custom hardware tends to maintain the utilization rate for Matrix-Vector computation as well, and thus the reduced computational complexity from skipping redundant decoding computation can be fully exploited. Also, note that the use of sparse matrix for computation in hardware can further reduce the overhead of weight load and make Matrix-Vector multiplication more efficient.

We notice that OpenNMT (Klein et al., 2017) also employs the concept of skipping redundant decoding computation in its Pytorch implementation. But the performance gain is limited for the reason we discussed above. In Section 7, we show that the impact of redundant computation skipping is much larger in the proposed custom accelerator than in GPU.

5 Set-Associative RCSC (SA-RCSC)

As explained in Section 2.2, the RCSC format (Park et al., 2019) is a sparse matrix format that mitigates problems with sparse matrix-vector multiplication (sM×dV) such as PE load imbalance and input load miss. While the RCSC format was originally proposed to increase the PE utilization for LSTM by exploiting unique characteristics of LSTM, it can actually be applied to any neural network in which an input is multiplied by multiple weight matrices. Since the Transformer also has such characteristics, we extend

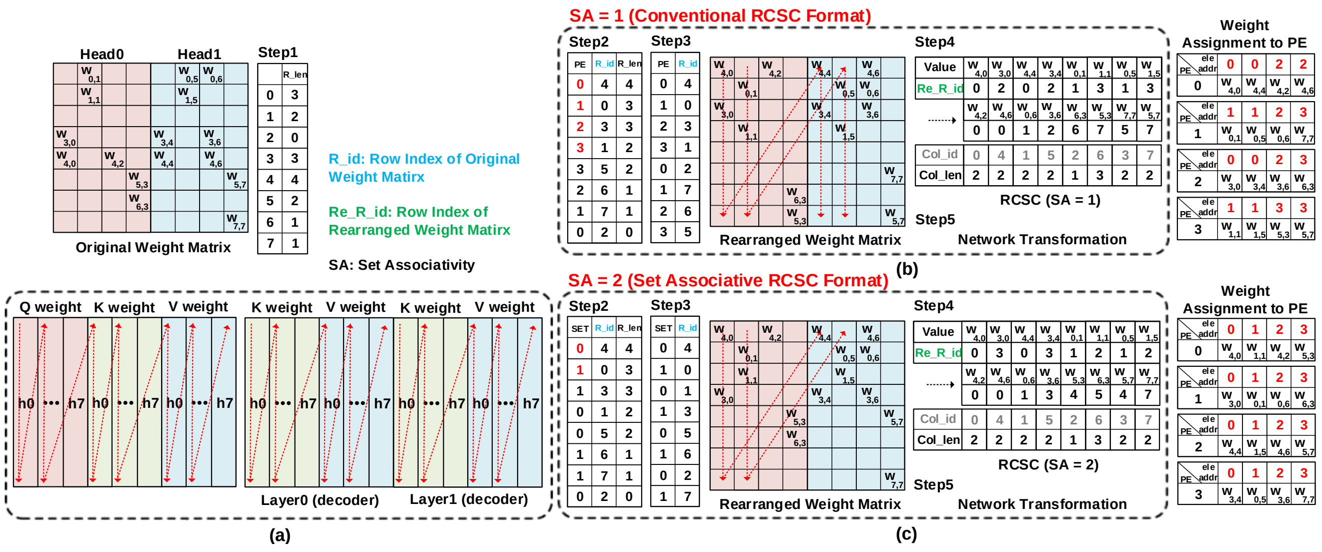


Figure 7. (a) The process of concatenating weights to apply the RCSC format. (b) The process of generating the conventional RCSC format (SA = 1). (c) The process of generating the proposed SA-RCSC format (SA = 2).

the RCSC format to express the sparse weight matrices of the Transformer. We also propose the SA-RCSC format to improve the PE utilization rate which tends to degrade when the original RCSC format is used for large number of PEs.

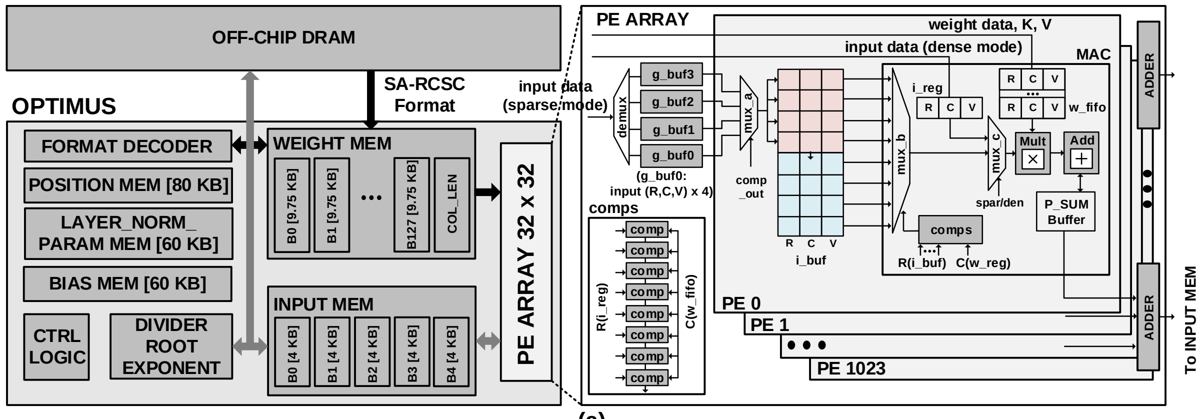
5.1 Generalizing RCSC for Transformer

The process of generating the RCSC format has two main goals. The first goal is to assign the non-zero values to the PEs evenly, so that the computational load of the PE is similar to each other (Step 2 in Fig. 1). The second goal is to reduce the input load miss by successively encoding the same columns of the weight matrices for all the gates which share the same input vector (Step 4 in Fig. 1). Note that the weight matrix (W^Q, W^K, W^V) of (masked) multi-head attention in the Transformer is also multiplied by the same input vector and there are 8 heads which share the same input, so that the locality of the loaded input vector is higher than that of LSTM.

5.2 SA-RCSC for Large-Scale PEs

In the conventional RCSC format, one PE is assigned to each row. If the number of PEs is much larger than the number of rows in the matrix, the number of rows processed by one PE becomes smaller. If the number of rows processed by one PE is too small, the locality of the input vector tends to become low as the locality becomes more sensitive to the distribution of non-zero elements in the row.

To mitigate this problem, we propose the SA-RCSC, in which a set of PEs instead of one PE is assigned to each row. With the proposed concept, the number of rows per set can be made relatively large so that the locality of input vector for the sets becomes higher. And, by assigning the weights to the PEs in a set alternately, the PEs in a set can have relatively high probability to share the same input vector. Let us show an example using a simple LSTM accelerator with four PEs (Fig. 7). Step 1 for the SA-RCSC is same as that of the conventional RCSC. The number of nonzero elements in each row is counted to assess the computation load. In step 2, the procedures for conventional RCSC and the proposed SA-RCSC start to differ. In conventional RCSC, four PE indices are sequentially assigned to the rows sorted in descending order of computation load (Step 2 in Fig. 7b). On the other hand, in SA-RCSC, only two set indices are sequentially assigned to the rows if the set associativity (SA) is 2 (Step 2 in Fig. 7c). If the SA is 4, only one set index would be assigned in the step 2. After the set indices (number of PEs in the accelerator / SA) are sequentially assigned from top rows, the next row with the largest number of non-zero values is assigned to the set index with the least computation load. In step 3 of the SA-RCSC, the pair of set index and row index is sorted so that the set index is to be in circular order to easily decode the set index assigned in the row. In step 4, the first column of eight heads of W^Q, W^K, W^V is successively encoded, and then the second column is sequentially generated in RCSC format. In step 5, network transformation is performed to keep the same output results regardless of rearrangement the weight matrix in step 3. Conventional RCSC and SA-RCSC formats are clearly distinguished when non-zero elements are assigned to PEs. In conventional RCSC, non-zero elements are assigned to PE according to the decoded PE index by modulo operation. In the table showing weight assignment to PE in Fig. 7b, $w_{4,0}$, $w_{4,4}$, $w_{4,2}$, $w_{4,6}$ are assigned to PE0. On the other hand, in SA-RCSC, non-zero elements with a decoded set index 0 are assigned to PE0 and PE2 alternately as they are in the same set. In the table showing weight assignment to PE in Fig. 7c, $w_{4,0}$, $w_{1,1}$, $w_{4,2}$, $w_{5,3}$ are assigned to PE0 and $w_{4,4}$, $w_{1,5}$, $w_{4,6}$, $w_{5,7}$ are assigned to PE2. Similarly, a non-zero



(a)

▭ Dense Matrix Multiplication ▭ Sparse Matrix Multiplication

State	State0	State1	State2	State3			State4			State5		
Data Fetch	Bias, Position Layer norm, x	W^Q,W^K,W^V	W^O	W^{F1}			W^{F2}			W^Q,W^K,W^V (for next layer)		
Data Size		$3\cdot d^2_{model}$	d^2_{model}	$2048\cdot d_{model}$			$d_{model}\cdot 2048$			$3\cdot d^2_{model}$		
Computation		x + Position = X	$[W^Q,W^K,W^V]\cdot X$ = Q,K,V	Softmax $(K^T\cdot Q/\sqrt{d_k}) = S$	$V\cdot S = Z^{0\text{-}7}$	$W^O\cdot Z^{0\text{-}7}$ = Z	X + Z = A0	Layer Norm (A0) = L0	ReLU $(W^{F1}\cdot L0 + B^{F1}) = F0$	$W^{F2}\cdot F0 + B^{F2} = F1$	A0 + F1 = A1	Layer Norm (A1) = L1
Number of Operations		$t\cdot d_{model}$	$3\cdot d^2_{model}\cdot t$	$d_{model}\cdot t^2$	$d_{model}\cdot t^2$	$d^2_{model}\cdot t$	$t\cdot d_{model}$	$d_{model}\cdot t$	$2048\cdot d_{model}\cdot t$	$2048\cdot d_{model}\cdot t$	$t\cdot d_{model}$	$d_{model}\cdot t$

(b)

Figure 8. (a) The overall architecture of OPTIMUS, a high-performance Transformer inference engine. (b) The control flow of the OPTIMUS. Dense matrix multiplication is colored in green, and a sparse matrix multiplication is colored in blue.

elements in set 1 are assigned to PE1 and PE3 alternately. In this example, the addresses of the input vector element required by PE0 is 0, 0, 2, 2 with conventional RCSC. In contrast, they are 0, 1, 2, 3 with SA-RCSC. As these addresses are requested sequentially, stalls due to input load miss decrease in the SA-RCSC case. Detailed experimental results for the PE utilization will be discussed in more detail in Section 7.2.

6 Proposed Hardware Architecture

6.1 Overall Architecture of OPTIMUS

The overall architecture of OPTIMUS, a customized system for high-performance Transformer inference, is shown in Fig. 8a.

PE array consists of N=1024 PEs, each of which is equipped with a MAC unit as well as internal buffers for temporarily staging in weight, input, and partial-sum data. A PE has two data paths to support matrix computation for both sparse and dense weights. In case of sparse weight (= sparse-mode), the hierarchical input buffer (g_buf and i_buf) (Park et al., 2019) is used to widen the search windows for input vector, thereby reducing the input load miss rate due to indexing sparse weights. In case of dense weight (= dense-mode), however, the hierarchical buffer is inefficient since it incurs unnecessary delay to fill it in with the shared input. Therefore, input in dense-mode streams into i_reg (instead of i_buf) to be directly multiplied with the dense weight. To support SA-RCSC, partial sums of PEs within a set are added via an adder tree. This across-PE accumulation is not needed for the conventional RCSC. See Appendix C for a detailed explanation of how OPTIMUS handles sparse and dense matrix multiplication.

OPTIMUS is also equipped with the shared data buffers for inputs and weights. WEIGHT MEM of 1.2MB (multi-banks of 4.8KB) is used to double-buffer weights as well as K, V matrix for skipping redundant decoding computation. Thanks to pruning, the requirement of WEIGHT MEM for double-buffering entire weights of a layer is reduced to 30% of the dense weight matrix (4MB). INPUT MEM also consists of multi-bank SRAMs to separately buffer input and partial-sums. Its size is set to stage-in at most 4-copies of input and partial-sums specifically targeting the single-batch use case of the decoder – four beams of input and partial-sums can be fully-kept in INPUT MEM so that one can avoid overhead of accessing DRAM to load/store them. This results in remarkable inference performance for the Transformer, as demonstrated in Section 7.

6.2 Supporting Diverse Matrix Computations

OPTIMUS is designed to achieve high performance for all kinds of matrix multiplications in the Transformer. In particular, OPTIMUS can achieve near-peak utilization for both matrix-matrix multiplication in the encoder and matrix-vector multiplication in the decoder with skipping redundant computations. In case of matrix-vector multiplication, SA-RCSC enables balanced parallelization of dot-product computations across the rows of weights, achieving high utilization even with a large number of PEs (N=1024). In case of matrix-matrix multiplication, OPTIMUS utilizes a customized dataflow to maximize weight reuse; weights loaded in WEIGHT-MEM is fully reused over all the partial-sums 1) across the samples in a batch, 2) across the time-steps (in the encoder) and 3) across the beams (in the decoder) via INPUT MEM and partial-sum buffers. Please refer to Fig. C.1 for more details on the dataflow.

The increase in this weight reuse comes at the cost of increased overhead of DRAM access for load/store of input/partial-sums. However, such overhead is relatively small compared to loading weights and K, V matrices. Note that the size of K and V matrices also increases over the increased weight reuse, but they are double-buffered along with the weight load, hiding its overhead behind the computation cycles. Together with the dedicate data paths for supporting sparse and dense weight matrices (as discussed in the previous section), OPTIMUS can achieve high utilization for the four different matrix computations of the Transformer.

6.3 Control Flow for Hiding Data Transfer Overhead

One of the key challenges in achieving high performance for the transformer inference is hiding the DRAM access overhead for its large model data. In OPTIMUS, we carefully designed a control-flow for double buffering (via finite state machines) to match the computation and data load cycles. As an example, Fig. 8b illustrates the weight fetch scheduling for a Multi-Head Attention layer. The computation sequence is grouped into 6 states, where each state is associated with a set of computations along with the weight to be prefetched in it. Note that the computation and the data load cycles can be estimated given a word-length; i.e., the computation cycle for COM1 = [$d^2_{model} \times t$] / [# MAC $\times$ Effective_PE_Util], whereas the data transfer cycle for Wo = [$d^2_{model} \times$Sparsity (dense=1.0)] / Bandwidth. By employing this cycle estimation and by considering the data dependency between the prefetched weight and the computations, we balanced the weight prefetch cycles and the compute cycles for all the states. As a result, we could measure that the spill-over cycles due to non-overlapped weight double-buffer was only 4.7% of the total computation.

7 Experimental Results

7.1 Experimental Setup

To evaluate the performance of OPTIMUS, WMT15 (EN-DE) (Sebastien Jean & Bengio, 2015), which is a representative benchmark data set for the Transformer, was used. For the evaluation of accuracy degradation due to pruning, the bilingual evaluation understudy (BLEU) (Papineni et al., 2002) score is used. We evaluated the latency and throughput of OPTIMUS as the average of 3200 sentences of different lengths. Since it takes too long to run such ex-

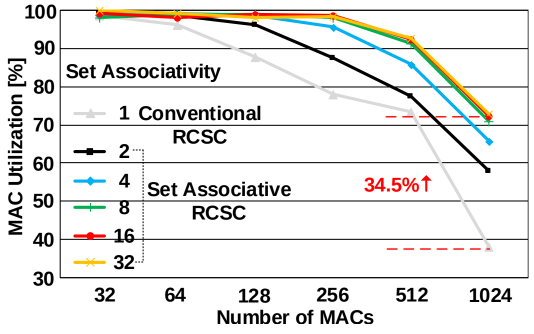


Figure 9. The MAC utilization for various number of MACs and set associativity (SA). The proposed SA-RCSC maintains very high MAC utilization rate even with the large number of MACs.

periments in RTL simulation, we devised a cycle-accurate simulation model, of which the cycle-by-cycle behavior is validated with the RTL simulation for the core PE block (including SA-RCSC-base data fetch, MAC operation, and partial-sum reduction). The precision for all the data used in MAC/layer-norm/softmax is 16-bit fixed-point, except for the accumulation in MAC (= 32-bit, then rounded). The row-index for SA-RCSC is 11-bit.

The weight matrix trained with PyTorch on the GPU was pruned using the well-known magnitude-based pruning to reduce the amount of data (Han et al., 2015b). The average pruning rate for all layers is 77.25%, which makes the amount of weight data stored in the SA-RCSC format 71.65% smaller than that of the dense matrix. The accuracy in terms of BLEU was decreased by 1.92% after the pruning. The detailed layer-by-layer description of the Transformer model and its pruned network is given in the Appendix D.

The hardware setup for running inference of the Transformer is as follows. The CPU result is measured from the inference using Intel(R) i7-6900K CPU @ 3.20GHz, and the GPU result is measured using NVIDIA Titan Xp with the latest CUDA kernel. The Neural Machine Translation Toolkit (Klein et al., 2017) is used for both CPU and GPU experiments. To the best of our knowledge, hardware accelerators dedicated for the Transformer neural network have not been reported yet. Thus, we design a custom Transformer hardware and apply CSC, RCSC and SA-RCSC formats to the weight data for the hardware to see the effects from different sparse matrix formats. Also, the redundant computation skipping is intentionally disabled/enabled to see the impact.

7.2 MAC Utilization

In accelerators which consist of large number of MACs, it is important to maintain high MAC utilization for small latency and high throughput. However, as mentioned in Section 3.2, a sparse matrix encoded in CSC and RCSC formats suffers from low utilization on large number of

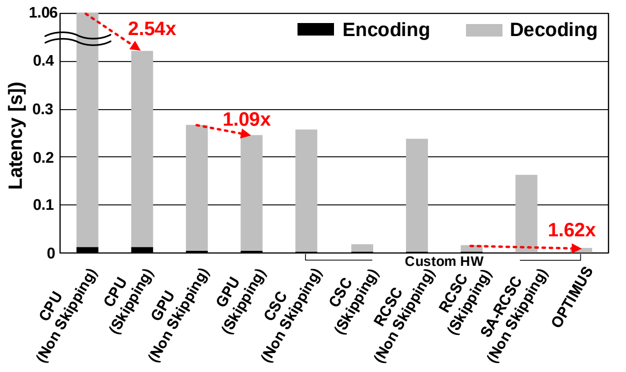


Figure 10. The inference latency of various hardware. The latency is measured for the average number of words ($t = 27$) for a batch size of 1 and a beam size of 4.

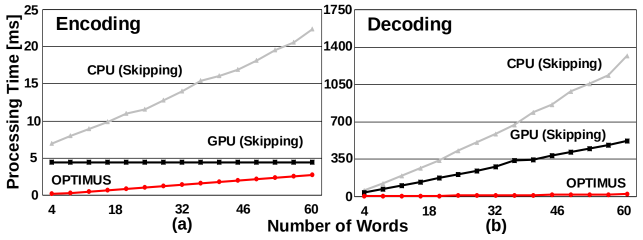


Figure 11. The processing time for (a) encoding and (b) decoding depending on the number of words on various hardware.

MACs. Simulation results confirm that the proposed SA-RCSC format maintains much higher MAC utilization when the number of MACs is large (Fig. 9). Note that, with 1024 MACs, SA-RCSC format with SA = 8 shows almost twice higher MAC utilization rate than that of the conventional RCSC (SA = 1 case). The MAC utilization increases as SA increases but it becomes saturated when SA $>$ 8 because the number of non-zero elements assigned to each PE starts to become relatively even at this condition.

7.3 Latency

In real-time processing applications, latency of single batch processing is one of the most important design parameters. As mentioned in Section 3, most of the computation time is spent on decoding because of the sequence to sequence structure (Fig. 10). The decoding processing time can be reduced by skipping redundant computations. The effect of redundant computation skipping varies from one hardware platform to another as mentioned in Section 4. With the skipping, the inference latency becomes 16.01$\times$ smaller in custom hardware, but the latency reductions are only 2.54$\times$ and 1.09$\times$ in the CPU and GPU, respectively (Fig. 10). In addition to the redundant computation skipping, the proposed SA-RCSC format gives additional 1.62 $\times$ reduction in latency thanks to the higher MAC utilization.

For encoding, GPU processing time could be shorter than OPTIMUS processing time when the number of words in one sentence is very large because GPU utilization can be

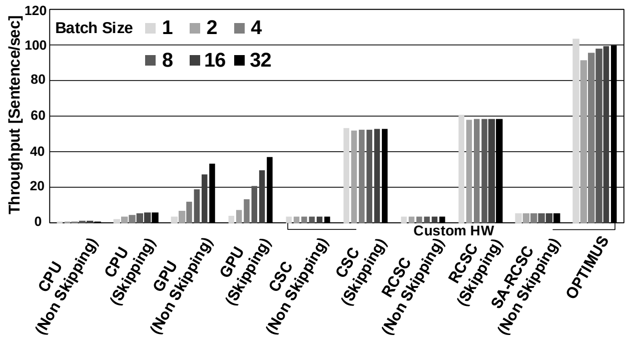


Figure 12. The throughput of various hardware for the batch size from 1 to 32.

maximized in the parallel encoding process (Fig. 11a). However, most of the computation time is spent on decoding, where the performance of the OPTIMUS is significantly better than that of GPU and CPU (Fig. 11b). In the decoding process, the processing time increases with the number of words in any hardware platform because of the iterative decoding characteristics. The performance gap between OPTIMUS and CPU/GPU becomes higher as the number of words increases thanks to the efficient vector-matrix multiplication in custom hardware which boosts up the effectiveness of redundant computation skipping.

7.4 Throughput

In server system or multi-user scenarios, the throughput analysis is important for batch sizes greater than 1. Fig. 12 shows the comparison of the throughput among CPU, GPU, and the proposed hardware. Here, the throughput is defined as the number of translated sentences per second (sentence/s), which is calculated by dividing the number of translated sentences by the processing time including DRAM access. Thanks to the combinations of weight pruning, SA-RCSC and computation skipping, processing time becomes highly short, so the throughput of the OPTIMUS is much higher than that of CPU and GPU for any batch size.

The throughput of GPU increases with the number of batches because the MAC utilization increases and weight data are reused as the batch size increases. On the other hand, the increase of throughput is relatively small in OPTIMUS case because the MAC utilization rate remains almost same regardless of the batch size. The modest throughput increase in OPTIMUS with the increased batch size mostly comes from the weight reuse in multi-batch scenario.

Note that OPTIMUS shows exceptionally high performance in the single batch case because we designed the hardware to keep all intermediate computation results local so that time-consuming DRAM access can be completely eliminated. This unique feature makes OPTIMUS an excellent candidate for real-time applications, where the latency of single batch inference is very important.

Table 2. Area and Power Consumption of OPTIMUS Core Blocks

COMPONENTS	AREA [μm^2]	POWER [mW]
TOP CONTROL	54348(1.05%)	10.43(1.42%)
MEMORY	2759794(53.21%)	57.24(7.82%)
G_BUF	1577(0.03%)	0.35(0.05%)
PERIPHERAL	23244(0.45%)	9.57(1.31%)
	1024 PEs	
CONTROL	325758(6.28%)	34.13(4.66%)
MACs	1930187(37.22%)	598.16(81.73%)
I_BUF	91294(1.76%)	21.96(3.00%)
TOTAL	5186201.416(100%)	731.84(100%)

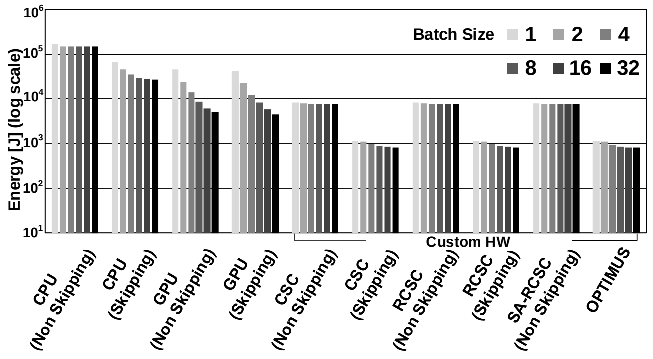


Figure 13. The energy consumption expressed in log-scale to process the test set (3200 sentences) for the batch size from 1 to 32.

Meanwhile, more widely used effective throughput (OPS) (Gao et al., 2018) is defined as the total number of operations to fully encode and decode a sentence divided by processing time. The effective throughput of OPTIMUS is 500.05 GOPS but we could not measure the OPS for CPU and GPU so direct comparison is not possible unlike the sentence/s metric.

7.5 Power Consumption and Energy Efficiency

For the power analysis, we synthesize OPTIMUS in a 28nm CMOS technology running at 200MHz with 1.0V. The area and power consumption of the on-chip components in OPTIMUS are extracted using Synopsys design compiler and the data are shown in Table 2. While the memory part occupies the largest area, power consumption is dominated by MACs as expected.

The CPU power measured by the likwid power meter (Treibig et al., 2010) is 50.46W, GPU power measured by the NVIDIA-SMI is 53.4W and the custom hardware consumes 731.84mW and DRAM power (196.3mW) was adopted from the Micron power calculator (Micron Technology, 2017). Total energy accounts for both accelerator and DRAM energy consumption. The energy consumed by a DRAM is calculated by multiplying the total amount of DRAM data access by the energy per unit bit (39 pJ/bit (Pawlowski, 2011)). There is orders-of-magnitude difference between the energy consumption in the OPTIMUS and CPU/GPU (Fig. 13). It is because the OPTIMUS finishes the inference operations much faster with smaller power. As the batch size increases, the energy tends to decrease on all hardware due to weight data reuse (Fig. 13). Although the largest energy reduction with the increased batch size is achieved on the GPU, OPTIMUS consumes the smallest energy for any batch size. Thanks to the high throughput and small energy consumption, the OPTIMUS shows 1464× and 155× higher energy efficiency (sentences/J) than GPU for a single batch case and a multi-batch case with batch size = 32, respectively (Fig. 14).

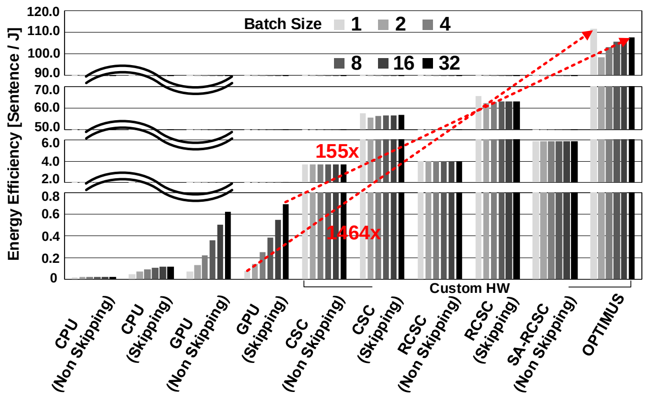


Figure 14. The energy efficiency of processing the test set (3200 sentences) for the batch size from 1 to 32.

8 CONCLUSION

This paper presents a custom hardware, OPTIMUS, for accelerating the Transformer neural network computation with high performance and high energy-efficiency. In order to run the inference efficiently, the encoding and decoding process were analyzed in detail, and dramatic performance improvement was achieved by skipping redundant computations in the decoding process. In addition, a SA-RCSC format was proposed to maintain high MAC utilization even when a large number of MACs are designed in the accelerator. These make latency, throughput, and energy efficiency of OPTIMUS much better than CPU, GPU and conventional custom hardware.

ACKNOWLEDGEMENTS

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ICT Consilience Creative program (IITP-2019-2011-1-00783) supervised by the IITP(Institute for Information & communications Technology Promotion).

REFERENCES

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *Inter-*

national Conference on Learning Representations (ICLR), 2015.

Cao, S., Zhang, C., Yao, Z., Xiao, W., Nie, L., Zhan, D., Liu, Y., Wu, M., and Zhang, L. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 63–72. ACM, 2019.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *CoRR, abs/1406.1078*, 2014.

Gao, C., Chang, et al. DeltaRNN: A power-efficient recurrent neural network accelerator. In *International Symposium Field-Programmable Gate Arrays (FPGA)*, pp. 21–30. ACM, 2018.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015a. URL http://arxiv.org/abs/1510.00149.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1135–1143, 2015b.

Han, S. et al. EIE: Efficient inference engine on compressed deep neural network. In *International Symposium Computer Architecture (ISCA)*, pp. 243–254, 2016. ISBN 978-1-4673-8947-1.

Han, S. et al. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *International symposium. Field-Programmable Gate Arrays (FPGA)*, pp. 75–84, 2017. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021745.

Klein, G., Kim, Y., Deng, Y., Senellart, J., and Rush, A. M. OpenNMT: Open-source toolkit for neural machine translation. In *Proc. ACL*, 2017. doi: 10.18653/v1/P17-4012. URL https://doi.org/10.18653/v1/P17-4012.

Micron Technology, I. Calculating memory power for ddr4 sdram. *Tech. Rep. TN-40-07*, 2017.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. BLEU: a method for automatic evaluation of machine translation. In *association for computational linguistics (ACL)*, pp. 311–318. Association for Computational Linguistics, 2002.

Park, J., Kung, J., Yi, W., and Kim, J.-J. Maximizing system performance by balancing computation loads in LSTM accelerators. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018. ISBN 978-3-9819263-0-9.

Park, J., Yi, W., Ahn, D., Kung, J., and Kim, J. Balancing computation loads and optimizing input vector loading in lstm accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019. ISSN 0278-0070. doi: 10.1109/TCAD.2019.2926482.

Pawlowski, J. T. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips Symposium (HCS)*, pp. 1–24, 2011.

Rizakis, M. et al. Approximate FPGA-based LSTMs under computation time constraints. In *International Symposium in Applied Reconfigurable Computing (ARC)*, 2018.

Sebastien Jean, Orhan Firat, K. C. R. M. and Bengio, Y. Montreal neural machine translation systems for WMT'15. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, 2015. URL https://www.aclweb.org/anthology/W15-3014.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Treibig, J., Hager, G., and Wellein, G. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pp. 207–216. IEEE, 2010.

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

Wang, S., Li, Z., Ding, C., Yuan, B., Qiu, Q., Wang, Y., and Liang, Y. C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, pp. 11–20. ACM, 2018.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016a.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016b.

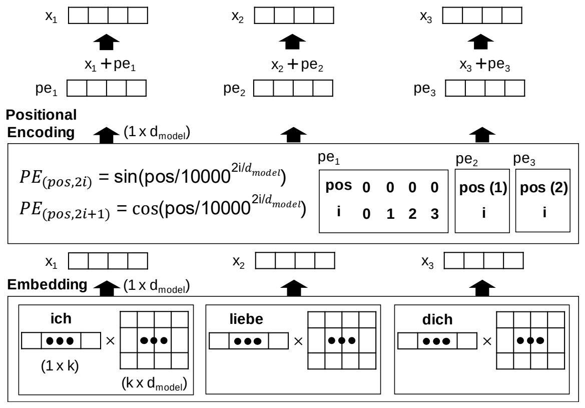

Positional Encoding
$PE_{(pos,2i)}$ = sin(pos/10000$^{2i/d_{model}}$)
$PE_{(pos,2i+1)}$ = cos(pos/10000$^{2i/d_{model}}$)
Embedding
ich
liebe
dich
(1 x k)
(k x d$_{model}$)
(1 x d$_{model}$)

Figure A.1. The process of embedding and positional encoding

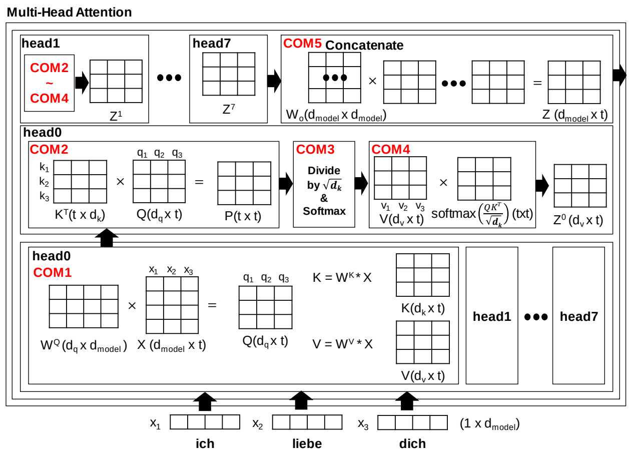


Figure A.2. The process of multi-head attention. This process is divided into five computations (COM1∼5).

A Transformer Computation Breakdown

A.1 Embedding & Positional Encoding

The first step of the Transformer is the word embedding (Fig. A.1). The words in a sentence are converted into vectors of d_{model} size through the embedding process. For example, a d_{model} size vector representing 'ich' is the result of multiplying the $d_{model} \times k$ size embedding matrix and one-hot k size vector representing 'ich', where k is the number of words that the embedding matrix can represent. Since the multiplied vector is one-hot vector, embedded word vectors can be computed by reading only the memory of the embedding matrix without multiplication.

Next, information about the relative or absolute position of each word should be injected into embedded word vectors, which is called positional encoding. Positional information is expressed through sine and cosine functions. The values of those functions are fixed values depending on the position of each element of a vector and the position of each vector in the sentence, so positional information can be referred to a lookup table without being computed every time. The vectors $(x_1, x_2, \cdots, x_{t_E})$ which is the summation of embedded word vectors and positional information are used as an input matrix ($d_{model} \times t_E$) of the encoder. This embedding and positional encoding process is also applied to output words when decoding.

A.2 Multi-Head Attention

Multi-head attention is the structure to measure the relationship among words in two same/different sentences. (Fig. A.2). This process is divided into five computations (COM1∼5). All computations except COM5 are progressed separately in the h heads which guarantee diverse attention maps for better translation quality.

The first computation (COM1) is a matrix-matrix multiplication that computes query (Q), key (K), and value (V). The size of the weight matrix (W^Q, W^K, W^V) is $(d_q, d_k, d_v) \times d_{model}$, where $d_q, d_k, d_v = d_{model}/h$. In the case of computing the COM1 in multi-head attention of the encoder and in masked multi-head attention of the decoder, the same input matrix is multiplied by W^Q, W^K, W^V to compute Q, K, V. On the other hand, when the COM1 in multi-head attention of the decoder is computed, K and V are computed by multiplying the final output of the encoder by W^K, W^V. Q is computed by multiplying the output of the masked multi-head attention of the decoder by W^Q. If W^Q, W^K, and W^V are pruned, COM1 becomes sparse matrix and dense matrix multiplication (sM×dM).

The second computation (COM2) is to compute the score. A score is computed as the inner product of K and V, which represents how words relate to each other. COM2 is always multiplication of two dense matrices (dM×dM) because any pruned weight is not used in COM2.

The third computation (COM3) is to divide the result of COM2 by the size of the key vector (d_k). This process scales down the value and stabilizes gradients during training (Vaswani et al., 2017). Through the softmax computation, all these values become positive and the element-wise sum in the query direction becomes always one.

The fourth computation (COM4) is to multiply the result of COM3 by value (V). This process reduces the information of unrelated words with low scores and increases that of words which need to be focused. Due to the same reason as COM2, COM4 consists of dM×dM.

The final fifth computation is to concatenate the results of COM4 (Z^0 - Z^7) in each head and multiply the concatenated results by the weight matrix (W^O) to mix them. If W^O is pruned, COM5 consists of sM×dM. After five computations

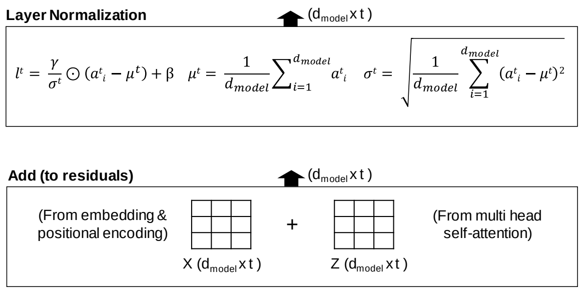

Layer Normalization
$(d_{model} \times t)$
$l^t = \frac{\gamma}{\sigma^t} \odot (a^t_i - \mu^t) + \beta$ $\mu^t = \frac{1}{d_{model}} \sum_{i=1}^{d_{model}} a^t_i$ $\sigma^t = \sqrt{\frac{1}{d_{model}} \sum_{i=1}^{d_{model}} (a^t_i - \mu^t)^2}$
Add (to residuals)
(From embedding & positional encoding)
X $(d_{model} \times t)$
Z $(d_{model} \times t)$
(From multi head self-attention)

Figure A.3. The process of the residual connection around each of the sub-layers, followed by layer normalization.

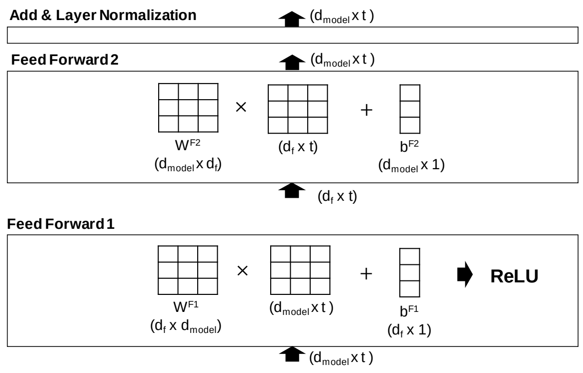

Add & Layer Normalization
Feed Forward 2
W^{F2} $(d_{model} \times d_f)$
$(d_f \times t)$
b^{F2} $(d_{model} \times 1)$
Feed Forward 1
W^{F1} $(d_f \times d_{model})$
$(d_{model} \times t)$
b^{F1} $(d_f \times 1)$
ReLU

Figure A.4. The process of position-wise feed forward network.

in multi-head attention, the output matrix still maintains the same size as that of the input matrix.

A.3 Residual Add & Layer Normalization

The output of the sub-layers in each encoder and decoder are added to their input, then the summation results are normalized in the layer-normalization process (Fig. A.3). The mean (μ^t) and standard deviation (σ^t) for the layer normalization are computed for each vector in the word direction. The normalized output is scaled by γ and is shifted by β, where γ and β is the trained parameters. This computation amount is much smaller than multi-head attention or position-wise feed forward (0.72% of total computations).

A.4 Position-wise Feed Forward

Each layer of the encoder and decoder has a fully connected feed-forward network. In this network, the input matrix is first linearly transformed after being multiplied by $W^{F1}[d_f \times d_{model}]$ and added by $b^{F1}[d_f]$, where d_f is the inner-layer dimension size. The first transformation result passes through Rectified Linear Unit (ReLU) activation, and the rectified result is linearly transformed again as the similar way to the first linear transformation. To maintain the dimension of the output by d_{model}, the size of weight W^{F2} and bias b^{F2} used in the second linear transformation should be $d_{model} \times d_f$ and d_{model} each. After W^{F1} and W^{F2} are pruned, the first transformation consists of sM×dM. On the other hand, the second one becomes multiplication between two sparse matrices (sM×sM), because its input matrix also has many zero values after passing through ReLU.

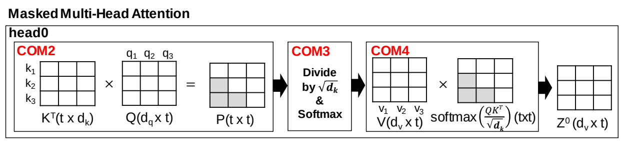


Figure A.5. The different computations (COM2 and COM4) of masked multi-head attention.

A.5 Masked Multi-Head Attention

Masked multi-head attention is additionally performed only at the decoder. This process is the same as the multi-head attention computation process except the computation of COM2 and COM4 (Fig. A.5). Unlike the correlation among all words in a sentence is computed in the encoder, the correlation between each word and its previous words is only computed in the masked multi-head attention. Therefore, after the correlation among all words is computed in COM2, the multiplication results between the queries of previous words and the keys such as $k_2 \times q_1$ and $k_3 \times q_2$ are masked as a negative infinity value to make those masked values converge to zero at COM3.

A.6 Linear & Softmax

The result of the multi-layer decoder process is converted into probabilities of all k words through a linear and softmax layer. A linear layer consisting of a fully-connected neural network projects the final output of the decoder into k-dimension. Note that k varies from dataset to dataset, and is usually as large as tens of thousands. Since the weight matrix size of the linear layer ($k \times d_{model}$) is very large, it is important to reduce the memory requirement of its weight matrix using pruning to reduce the amount of computations.

The softmax layer converts the output of the linear layer into a probability matrix of all k words. The word with the highest probability is selected as the final result of that decoding step. In the inference process, because only the word with the highest score is selected, the softmax process can be skipped.

A.7 Beam Search

The most common way to search a target sentence is to select the word which has the highest probability for every decoding-step. This way is based on the greedy algorithm, however, is not guaranteed whether this method always generates a best target sentence. The beam search supple-

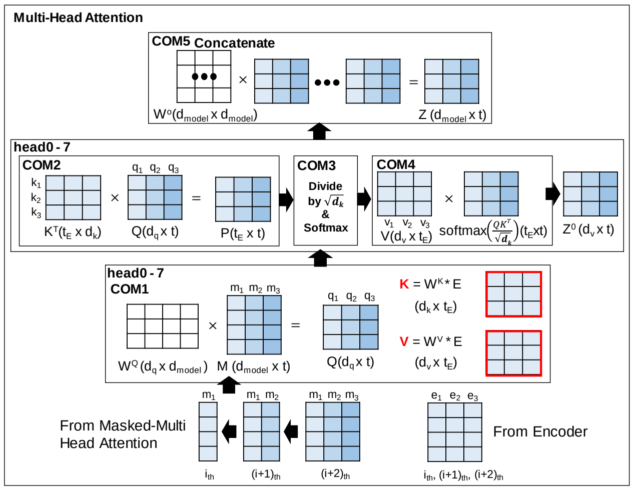


Figure B.1. Analysis of redundant decoding computations of multi-head attention

ments the limitation of the greedy search. In the beam search method, the sentences where their cumulative probability for each word falls within top-n are selected for each decoding-step, where n is the beam size. Note that the beam search is as the same method as the greedy search algorithm when $n = 1$. This beam search increases the translation performance of a neural machine translation model, however, more resources and computation power are required because the input size of the model is increased by n.

B Skipping Redundant Computations of Multi-Head Attention in Decoders

As mentioned in Section A.2, K and V in the multi-head attention of the decoder are computed by using the final output of the encoder (Fig.B.1). That is, K and V are fixed matrices once they are computed at the first decoding time-step. We can skip the computations for K and V for other decoding time-steps by loading/storing the computed K and V. Furthermore, due to the fixed K and V, z_t which is the vector element of Z at time-step t is only dependent on q_t, the query at time-step t. This property allows skipping redundant decoding computations to be applied to even multi-head attention in the decoder layers. In summary, only the vector from the output word of the previous decoding time-step is required as an input of the decoder for each decoding time-step.

C Sparse/Dense Matrix Computation Flows in OPTIMUS

In this section, we describe the details of the computation flows in OPTIMUS, focusing on the matrix multiplication flows. The sM×sM multiplication for the sparse weight and the sparse input matrix is done as follows. Tiny sized g_buf and i_buf are assumed for a simple example and the exemplary sparse weight matrix and input matrix are shown in Fig. C.1. Note that the number of input matrix columns that can be loaded in OPTIMUS depends on the size of the P_SUM buffer in the MAC. The example assumes that inputs for up to two time steps can be stored, so the inputs for t_0, t_1 are loaded into i_buf via g_buf from INPUT_MEM in the order $a_{0,0}$, $a_{0,1}$, $a_{1,1}$. The sparse weight matrix is encoded in SA-RCSC format and loaded via w_fifo. Then, the column index of the weight element and the row index of the input vector element are compared in the comparators (comps), and if they match, the input value is multiplied by the weight value, so $w_{0,0}$ and $a_{0,0}$ are multiplied in this example. This value is stored in the P_SUM buffer and is added to the result of other dot product with the same index information. Since the column index of $w_{0,0}$ and the row index of $a_{0,1}$ also matches, $w_{0,0} \cdot a_{0,1}$ is executed. When there are no more input elements that match the column index of $w_{0,0}$, the pointer of w_fifo points to $w_{2,1}$. Similarly, the column index of $w_{2,1}$ is compared with the row of i_buf. When $a_{1,1}$ is matched, the value in the red region in i_buf is shifted to the blue region and two input elements are newly loaded from g_buf. This control method minimizes the occurrence of stalls because the larger search window allows the input elements to be prepared even if the address of the requested input elements is irregular due to sparse weight. After the sixth computation shown in the computation order in the Fig. C.1, the MAC computations for t_0 and t_1 are completed. The value stored in the P_SUM buffer is added to the value in the P_SUM buffer of another PE with the same SA number and then it is stored in INPUT_MEM. If there are no more tokens to be computed other than t_0, t_1, the tokens are directly used as an input of the next computation. However, if word length exceeds the internal P_SUM buffer size, the value of t_0, t_1 are stored in the DRAM and the weight matrix must be reloaded to compute t_2, t_3.

The process for multiplication of dense weight matrix with dense input matrix is simpler than the process for the sparse matrix computation. The order of input matrix loading is the same as that for the sparse input case. However, input elements are loaded through i_reg rather than g_buf and i_buf. Unlike the sparse matrix computation where all PEs are loaded with the same input data, different input values are loaded to each PE in the dense matrix computation case. Therefore, the hierarchical buffer structure is not suitable for each PE to load input vector element separately because the input vector elements are shared by all PEs when the hierarchical buffer is used. The parts where the dense matrix multiplications are performed are the COM2 and COM4 process of the masked multi-head attention and the multi-head attention. In these processes, the row size of the weight

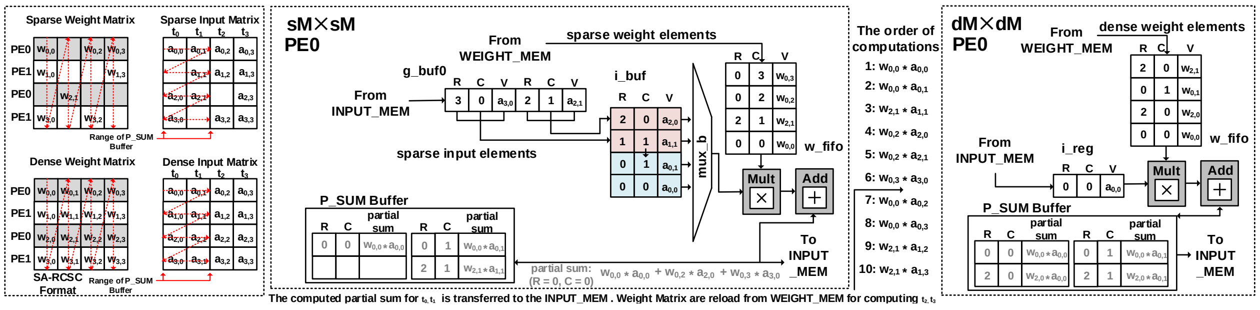


Figure C.1. Detailed description of how sM×sM and dM×dM are computed inside a PE of OPTIMUS.

matrix is t or d_{model} (Fig. 5). This row size is smaller than that of the weight matrix where the sparse matrix computation is performed. So one PE processes fewer rows than sparse matrix multiplication case so that the partial sum for more columns of input matrix can be accumulated in the P_SUM buffer. As a result, high reuse of weight data can be achieved. Since the weight matrix is not sparse, the value is transferred to the w_fifo in the column direction without using the sparse matrix format. The pointer of w_fifo is shifted every cycle and the calculated P_SUM buffer value is transferred to the INPUT MEM similar to the sparse matrix multiplication case.

D Pruning Results of the Transformer Model

We first trained a 6-layer transformer model with $h = 8$, $d_{model} = 512$, $d_f = 2048$, and $n = 36549$ using WMT English-to-German (EN-DE) dataset (Sebastien Jean & Bengio, 2015) under the same training condition as suggested in (Klein et al., 2017) and (Vaswani et al., 2017). After finishing training, we pruned the weights in the transformer model with the pruning rates shown in Table D.1 using the magnitude-based pruning method (Han et al., 2015b). Then we retrained the pruned model while maintaining the above training condition except the learning rate schedule; we use the learning schedule scaled by 1.25 compared to the original one. The weights of the transformer model are removed by 77.25% in average, but the BLEU score only degrades about 0.6 in the WMT15 EN-DE dataset (Table D.1).

Table D.1. The sparsity of the pruned Transformer model and BLEU evaluation results on WMT15

Layer	Sub Layer	Matrix Size	Pruning Rate [%]	Data Size (Dense) [KB]	Data Size (Pruned) [KB]	BLEU
Encoder0	MHA	512x512	77.93	2048	567.27	Pre Pruning: 32.29 Post Pruning: 31.67
	FF	2048x512	73.39	4096	1368.21	
Encoder1	MHA	512x512	77.89	2048	586.14	
	FF	2048x512	75.12	4096	1279.68	
Encoder2	MHA	512x512	77.92	2048	567.56	
	FF	2048x512	75.18	4096	1276.77	
Encoder3	MHA	512x512	78.02	2048	565.00	
	FF	2048x512	75.26	4096	1272.52	
Encoder4	MHA	512x512	77.97	2048	566.15	
	FF	2048x512	75.31	4096	1270.09	
Encoder5	MHA	512x512	77.91	2048	567.73	
	FF	2048x512	75.17	4096	1277.11	
Decoder0	MMHA	512x512	78.09	2048	563.04	
	MHA	512x512	77.99	2048	565.68	
	FF	2048x512	75.08	4096	1281.80	
Decoder1	MMHA	512x512	77.99	2048	565.59	
	MHA	512x512	78.09	2048	563.06	
	FF	2048x512	75.06	4096	1282.88	
Decoder2	MMHA	512x512	78.01	2048	565.77	
	MHA	512x512	77.97	2048	566.28	
	FF	2048x512	74.99	4096	1286.58	
Decoder3	MMHA	512x512	78.00	2048	565.52	
	MHA	512x512	77.95	2048	566.77	
	FF	2048x512	74.96	4096	1288.27	
Decoder4	MMHA	512x512	78.02	2048	564.87	
	MHA	512x512	77.97	2048	566.10	
	FF	2048x512	75.02	4096	1284.88	
Decoder5	MMHA	512x512	77.99	2048	565.60	
	MHA	512x512	77.90	2048	567.95	
	FF	2048x512	75.04	4096	1284.03	
Linear		36549x512	79.77	36549	9104.25	

MMHA: Masked Multi-Head Attention, MHA: Multi-Head Attention, FF: Position-wise Feed Forward