University of Amsterdam

Parallel Computing Systems Group, Informatics Institute
Master of Science in Computer Science

Performance Engineering of Transformer Inference on Edge Device

Candidate: Xu Hang
Supervisor: Dr. Anuj Pathania
Second Reader: Dr. Semeen Rehman
Date: August, 2024

Abstract

Transformer models have initiated a renaissance in machine learning due to their state-of-the-art performance in handling natural language processing (NLP) tasks. The Transformer is the most complex neural network to date, but the increasing model size makes it cumbersome to run on resource-constrained platforms such as edge devices. Modern edge devices employ Heterogeneous Multi-Processor System-on-Chips (HMPSoCs), incorporating processors with different characteristics. Understanding Transformer model inference characteristics on different hardware is crucial for optimizing runtime resources and designing efficient edge AI systems.

This work developed the ARM Computing Library (ARM-CL) to support Transformer inference on both CPU and GPU in edge devices, achieving a 3x faster inference than TVM implementations. This paper further studied the characteristics of Transformer layers during inference on CPU and GPU, examining model inference latency under different workloads and model parameter settings. Results show that Transformer layers exhibit distinctive behaviors, Scaled Dot-Product Attention (SDPA) layer takes up more than 90% of GPU inference latency, and feed-forward (FF) layer computes more than 900 times faster on GPU than CPU.

Contents

1

Introduction

Enabled by the synergy between large datasets and extensive computing power, the current trend in designing Neural Network (NN) models is going deeper in architecture and using larger parameter sets for higher accuracy [1]. This results in models that are harder to train and more computationally resource-intensive, both during training and inference.

While model training is only feasible on computer clusters for modern NN models, model inference has real-world applications on edge devices such as object detection [2] and semantic segmentation [3]. As edge devices are often resource-constrained, there is a growing interest in balance model inference accuracy and computational performance for efficient NN inference to be conducted on edge devices.

Transformer models [4] are gaining popularity for their state-of-the-art performance in handling NLP tasks. The Transformer opens up a new research direction as its architecture combines structure from Convolution Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). It fully deprecates recurrence and relies entirely on attention mechanism, featuring more operation types. They are primarily composed of compute-intensive matrix multiplications (MMULs) and memory-intensive non-linear operations. In addition to having a more complex data flow, Transformers represent the most complex architecture to date.

To improve model inference performance, understanding the hardware characteristics of target platforms is crucial. Modern system-on-chip designs often integrate multiple processors on the same semiconductor die. Furthermore, it is commonplace to have CPU and GPU within HMPSoC [5]. To utilize computing power from HMPSoC, various computing frameworks[6]–[8] and compilers [9] have been introduced. Such tools transform high-level NN model descriptions into concrete operations to be executed on target hardware. This enables the efficient deployment of NN models.

However, existing efforts to improve NN performance on edge devices have primarily focused on CNNs and RNNs. Existing studies on Transformer models focused on architecture that impacts inference accuracy[10], [11]. However, there is a gap in understanding the runtime characteristics of Transformer architectures. Since NN models are not monolithic, different layers within an NN model exhibit distinctive performance characteristics on both CPU and GPU. Investigating layer behavior can aid in the design of more efficient edge AI and runtime execution. This work yields the following novel contributions:

1. The first work developed ARM-CL to support Transformer model inference, providing optimized performance on ARM architecture devices.
2. Further, it provided data and analysis of the workload characteristics of Transformer model running on real-world embedded platforms.

1.1. Motivation

We chose BERT as a Transformer implementation because it achieved state-of-the-art performance on a variety of NLP tasks and can be used for various downstream applications, such as text classification and question-answering. BERT is better suited for edge computing compared to GPT [12], because GPT generates sentences in a chatbot-like manner. ARM-CL on the other hand provides direct control over layer runtime behavior and offers more optimized performance compared to TVM. This allows for more accurate measurement of model characteristics.

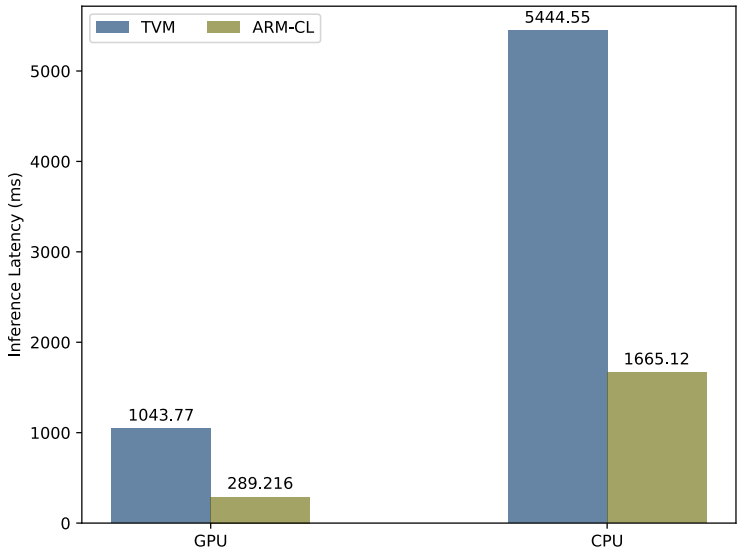

Figure 1.1: BERT base model inference latency comparison between TVM and ARM-CL on Khadas Vim 3.

1.2. Related Work

There are optimizations from both hardware and software perspectives to improve edge AI inference. From the hardware perspective, one approach is to generate specialized Application-Specific Integrated Circuits (ASICs) in order to provide acceleration [13]. Work trying to improve edge AI inference from a software perspective ranges from optimizing model architecture to improving runtime behavior. This paper will focus on software optimizations, which are more applicable to general processors.

To build efficient NNs, research suggests improving the model architectural design by replacing the original micro-architecture with more efficient and specialized micro-architectures. For instance, the inception module [14] was originally designed for CNNs and replaces the

original sparse convolution structure with dense components to reduce computation. This has been adopted in MobileBERT [15] to build a narrow version of BERT while maintaining trainability, being 4.3x smaller and 5.5× faster than BERT-base on GLUE [16] benchmark.

A different perspective suggests utilizing macro-architecture optimization techniques that explore layer structures and connectivity designs within NN models, such as in 'Group convolution' proposed in ShuffleNet [17]. It has been applied in SqueezeBERT [18]. This achieved 4.3x lower inference latency on mobile devices compared to BERT-base while maintaining competitive accuracy on the GLUE benchmark.

Further work explores model compression, which compresses existing models into smaller versions in a lossy manner using methods such as network pruning and data quantization [19]. Network pruning involves reducing the structure of NNs by removing redundant connections or layers. Data quantization reduces model size by decreasing the number of bits used to represent each weight. Work by Michel et al. [20] studies multi-head attention layers from a network pruning perspective and demonstrates that pruning certain attention heads does not significantly degrade model accuracy. Knowledge distillation [21], which typically uses a larger teacher model to train a smaller student model, transfers knowledge from the large model to the smaller one and allows the former to replicate the behavior of the latter. DistilBERT [22] uses knowledge distillation to replicate the behavior of a larger model, resulting in a smaller model with fewer layers (40% fewer parameters; 60% faster runtime). Despite its reduced size, it retains 97% of the performance of original model on benchmarks.

Complex model architectures offer numerous design options, such as the number of layers and the size of convolution kernels, thereby expanding the model design space. Several hyper-parameter search methods have been proposed to automatically explore these options. These methods include Bayesian optimization [23], genetic algorithms [24], or reinforcement learning-based Neural Architecture Search (NAS) [25]. NAS has attracted great interest from academia, and various works [26]–[28] have applied NAS to build efficient Transformers.

Approaches to improve model runtime execution are carried out in a hardware-aware manner, aiming to fully utilize the resources offered by the target hardware. Parallelism is often exploited to maximize HMPSoC hardware utilization. Data-level parallelism approaches, such as μLayer [29], partition the execution of a single NN layer across different processors to minimize single inference latency. Task-level parallelism, like pipelining [30], [31], executes different parts of network partitions from a batch of tasks simultaneously on different processors to improve overall inference throughput. However, there is no existing work supporting Transformer inference using ARM-CL.

2

Background

2.1. BERT Architecture

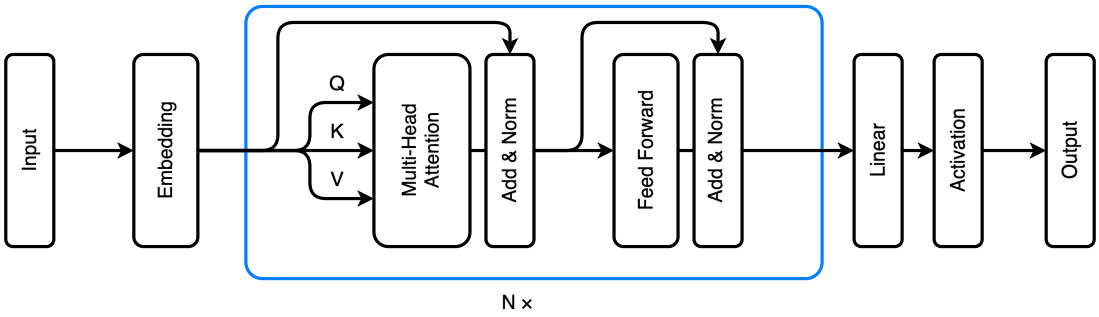

Figure 2.1: BERT architecture. Contains only encoder part of Transformer.

BERT (Bidirectional Encoder Representations from Transformers) [32] model architecture consists of $N = 12$ transformer encoders stacked on top of each other. Different from CNNs and RNNs which process images, Transformers are used for NLP processing which takes sentence input. Sentence input is first transformed into token representation at input layer. The output is then represented as numerical tensor after embedding layers. Embedding layers contain token embedding, segment embedding and positional embedding sub-layers. Each encoder module includes a multi-head attention (MHA) layer and a FF layer. Input to MHA layer is then used as query, keys and values to compute SDPA:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

After MHA layer follows a fully connected FF layer, to allow the different attention heads to interact with each other. Both sub-layers employ residual connection [33] around input and output. Output from two sub-layers is first element-wise added with input, then layer normalization is performed [34] on the added product. Output from the last encoder is followed by a linear layer and an activation layer for post-classification.

2.2. ARM Computing Library

NN inference usually follows a general framework transforming high-level NN model description into runtime execution: NN models are described using a directed graph, where nodes represent instances of NN operations and connections represent data flow within the NN structure. An operation represents an abstract computation, such as a convolution or MMUL. Each operation wraps over one or more kernels, which is a specific implementation of the computation task to be run on a particular target processor[6].

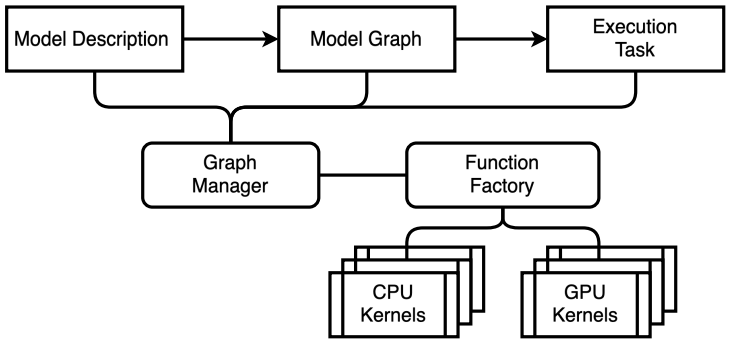

Figure 2.2: Architecture of ARM-CL execution configuration workflow.

ARM-CL follows this framework and provides a set of optimized kernels. It utilizes NEON vectorization for CPU acceleration and OpenCL for GPU acceleration [8]. The model description outlines the model layer structure, which the graph manager translates into a model graph. The function factory then converts the model graph into a sequence of target-specific execution tasks during configuration (Listing 2.1). Tensor memory and the execution of configured tasks are managed by the runtime backend.

Listing 2.1: Function Factory

```
std::unique_ptr<IFunction> FunctionFactory::create(INode *node, GraphContext &ctx
    )
{
    switch (node->type())
    {
        case NodeType::ActivationLayer:
            return detail::create_activation_layer<ActivationLayer, TargetInfo>(
                *polymorphic_downcast<ActivationLayerNode *>(node));

        ...

        case NodeType::ScaleDotProductionAttentionLayer:
            return detail::create_scale_dot_production_layer<
                                ScaleDotProductionAttentionLayer,TargetInfo>(
                *polymorphic_downcast<ScaleDotProductionAttentionNode *>(node));
        case NodeType::LayerNormLayer:
            return detail::create_layer_norm_layer<LayerNormLayer, TargetInfo>(
                *polymorphic_downcast<LayerNormNode *>(node));
        default:
            return nullptr;
    }
}
```

3

Inference with ARM-CL

3.1. Model Description

This work used the pre-trained base-uncased model by Devlin et al., with $N = 12$ layers and $h = 12$ attention heads [32]. This work implemented three kernels—embedding, linear, and layer normalization—as these kernels are missing from the original ARM-CL, along with related operations to support BERT inference. The implementation of the ARM-CL BERT model layer description is provided in Listing 3.1:

Listing 3.1: BERT model ARM-CL description

```

class GraphBERT
{
    public:
    bool do_setup(int argc, char **argv)
    {
        graph << InputLayer(get_token_accessor(common_params),
                            get_segment_accessor(common_params))

              << EmbeddingLayer(get_weights_accessor("token_embedding.npy"),
                                get_weights_accessor("segment_embedding.npy"),
                                get_weights_accessor("positional_embedding.npy");

            add_encoder_block("layer_0/");
            add_encoder_block("layer_1/");
            add_encoder_block("layer_2/");
            add_encoder_block("layer_3/");
            add_encoder_block("layer_4/");
            add_encoder_block("layer_5/");

            add_encoder_block("layer_6/");
            add_encoder_block("layer_7/");
            add_encoder_block("layer_8/");
            add_encoder_block("layer_9/");
            add_encoder_block("layer_10/");
            add_encoder_block("layer_11/");

        graph << LinearLayer(get_weights_accessor("pooler_weight.npy"),
```

```
                                get_weights_accessor("pooler_bias.npy"))

                <<ActivationLayer(ActivationLayerInfo(ActivationFunction::TANH))

                << OutputLayer(get_output_accessor(common_params));

        graph.finalize(common_params.target, config);

        return true;
    }
};
```

ARM-CL uses a directed graph instance to describe the connectives between layers. Starting with an input layer followed by an embedding layer that loads pre-trained token, segment, and positional embedding. It then adds 12 encoder blocks (Depicts in Listing 3.2) as BERT base-uncased model has 12 encoder blocks. Then the model concludes with a linear layer and a Tanh activation layer for the pooling operation. In addition, an output layer. Finally the graph manager is called to configure execution task using the provided target and configuration parameters.

The 'add_encoder_block' function defines an encoder block as shown in Figure 2.1. It creates two sub-streams for residual connection, one with MHA layer and one without MHA. Result after MHA and original residual input is element-wise added followed by layer normalization. MHA layer is implemented using two sub-layers, 'AttentionLinearLayer' contains three linear operations and loads corresponding model weight, 'ScaleDotProductionLayer' computes scale dot production. FF layer also adopted sub-stream implementation for residual connection, it contains two linear layers and a GELU activation in between. Result is later element-wise added and layer normalized.

Listing 3.2: ARM-CL add BERT encoder block to graph

```
void add_encoder_block(std::string layer_path)
{
    SubStream without_attention(graph);
    SubStream with_attention(graph);

    with_attention
        << AttentionLinearLayer(
                        get_weights_accessor(layer_path, "query_weight.npy"),
                        get_weights_accessor(layer_path, "query_bias.npy"),
                        get_weights_accessor(layer_path, "key_weight.npy"),
                        get_weights_accessor(layer_path, "key_bias.npy"),
                        get_weights_accessor(layer_path, "value_weight.npy"),
                        get_weights_accessor(layer_path, "value_bias.npy"))

        << ScaleDotProductionLayer();

    graph << EltwiseLayer(with_attention,without_attention,
                          EltwiseOperation::Add);

    graph << LayerNormLayer();

    SubStream without_ff(graph);
    SubStream with_ff(graph);
```

```
24
25      with_ff << LinearLayer(
26                          get_weights_accessor(layer_path, "ff_weight_0.npy"),
27                          get_weights_accessor(layer_path, "ff_bias_0.npy"))
28
29              << ActivationLayer(ActivationLayerInfo(ActivationFunction::GELU))
30
31              << LinearLayer(
32                          get_weights_accessor(layer_path, "ff_weight_1.npy"),
33                          get_weights_accessor(layer_path, "ff_bias_1.npy"));
34
35      graph << EltwiseLayer(with_ff, without_ff, EltwiseOperation::Add);
36
37      graph << LayerNormLayer();
38  }
```

3.2. Sentence Input Embedding

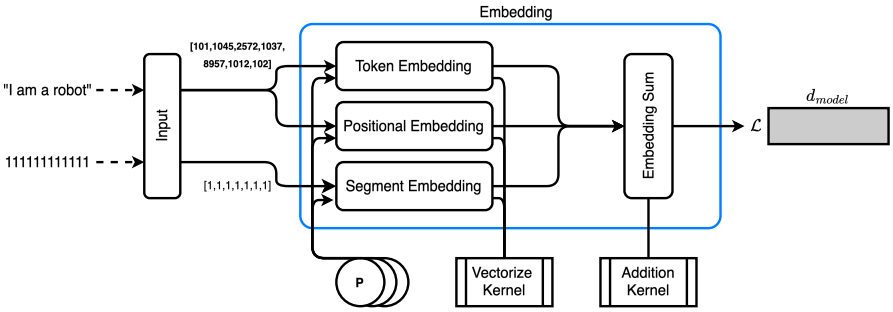

Figure 3.1: Input layer takes sentence and segment input, producing $\mathcal{L}$ scalars for the embedding layer. P denotes pre-trained model parameters, including weight and bias. The example output tensor has a shape of (7, 768).

Input to the Transformer model includes a sentence input, a segment input, and optionally a mask input. The input sentence is first translated into numerical token representation at the input node. This work employs the longest matching algorithm to tokenize the input sentence and uses a pre-trained WordPiece vocabulary [35] from the original BERT implementation to convert word tokens into numerical representations. BERT processes a maximum token length of 512 for a single inference. Since sentence inputs can have arbitrary lengths, padding is often added to reach the input length of 512. To save computation, this work does not add extra padding, allowing the token input to have an arbitrary length of $\mathcal{L}$.

The embedding layer consists of four operations: token embedding, segment embedding, positional embedding, and finally an element-wise addition to sum the previous three outputs, as shown in Figure 3.1. Original ARM-CL only supports image processing; therefore, this work developed the entire embedding layer and vectorize kernel to support sentence input. Vectorize kernel maps a single scalar input to a vector of d_{model} length by copying values from the loaded model parameter tensors. The loaded token embedding vocabulary parameter is a 2D tensor with the shape (30522, 768). Each input token scalar is mapped to a feature vector row from this vocabulary tensor.

The three embedding operations load the corresponding pre-trained model parameters, map

each scalar input to numerical vector representations and produce three 2D tensors with the shape ($\mathcal{L}$, d_{model}). Finally, after the embedding sum layer, the embedding layer output is a 2D tensor with shape ($\mathcal{L}$, d_{model}).

Listing 3.3 shows transforming model layer description into graph node description, each node represents one operation. It begins by obtaining the tensor descriptor for the input tensor, then defines tensor shapes for embedding layer nodes. Subsequently, it adds constant nodes to the graph for the vocabulary, segment, and position embedding model weights, each associated with its passed-in accessor. The function then creates nodes for corresponding layers and connects them to their respective inputs and constant nodes.

Listing 3.3: Create embedding layer nodes

```
NodeID GraphBuilder::add_embedding_node(Graph &g, NodeIdxPair input,
                                        EmbeddingLayerInfo emb_info,
                                        ITensorAccessorUPtr     vocabs_accessor,
                                        ITensorAccessorUPtr     segemnts_accessor,
                                        ITensorAccessorUPtr     position_accessor)
{
    const TensorDescriptor input_tensor_desc =
                        get_tensor_descriptor(g, g.node(input.node_id)->outputs()[0])
                            ;

    TensorDescriptor v_desc = input_tensor_desc;
    v_desc.shape = TensorShape(emb_info.d_model(),emb_info.d_vocab());
    TensorDescriptor s_desc = input_tensor_desc;
    s_desc.shape = TensorShape(emb_info.d_model(),emb_info.d_segment());
    TensorDescriptor p_desc = input_tensor_desc;
    p_desc.shape = TensorShape(emb_info.d_model(),emb_info.d_position());

    NodeID v_c_nid  = add_const_node_with_name(g, "vocabs", v_desc,
                                                        std::move(vocabs_accessor
        ));
    NodeID s_c_nid  = add_const_node_with_name(g, "segements", s_desc,
                                                      std::move(segemnts_accessor)
        );
    NodeID p_c_nid  = add_const_node_with_name(g, "position", p_desc,
                                                       std::move(position_accessor
        ));

    NodeID t_nid = g.add_node<TokenEmbeddingLayerNode>();
    g.add_connection(input.node_id, 0 /* text input*/, t_nid, 0);
    g.add_connection(v_c_nid, 0, t_nid, 1);
    NodeID s_nid = g.add_node<SegmentEmbeddingLayerNode>();
    g.add_connection(input.node_id, 1 /* segment input*/, s_nid, 0);
    g.add_connection(s_c_nid, 0, s_nid, 1);
    NodeID p_nid = g.add_node<PositionEmbeddingLayerNode>();
    g.add_connection(input.node_id, 0 /* text input*/, p_nid, 0);
    g.add_connection(p_c_nid, 0, p_nid, 1);

    NodeID add_nid = g.add_node<EmbeddingSumLayerNode>(emb_info);
    g.add_connection(t_nid, 0, add_nid, 0);
    g.add_connection(s_nid, 0, add_nid, 1);
    g.add_connection(p_nid, 0, add_nid, 2);
    return add_nid;
}
```

3.2.1. Vectorize Kernel

Listing 3.4 shows the CPU implementation for vectorize kernel. It maps scalar values from sentence token tensor (src) to corresponding vector from model weight tensor (vector), and stores the results in output tensor (dst). It iterates over the pre-configured execution window range and populates output tensor using 'memcpy'.

Listing 3.4: CPU Vectorize Kernel

```
void neon_vectorize_int_2_float32(const ITensor *src,const ITensor *vector,
                                  ITensor *dst, const Window &window)
{
    Window win(window);

    const unsigned int window_start_x   = win.x().start();
    const unsigned int window_end_x     = win.x().end();

    const unsigned int vector_depth     = vector->info()->tensor_shape().x();

    unsigned int offset_vector,offset_dst;

    win.set(Window::DimX, Window::Dimension(0,1,1));

    Iterator src_iter(src,win);
    Iterator dst_iter(dst,win);
    Iterator vector_iter(vector,win);

    const auto src_ptr      = reinterpret_cast<unsigned int *>(src_iter.ptr());
    const auto dst_ptr      = reinterpret_cast<float *>(dst_iter.ptr());
    const auto vector_ptr   = reinterpret_cast<float *>(vector_iter.ptr());

    execute_window_loop(win,
        [&](const Coordinates &)
        {
            for(unsigned int x = window_start_x; x < window_end_x; x++)
            {
                offset_dst     = x * vector_depth;
                offset_vector  = *(src_ptr+x) * vector_depth;
                std::memcpy(dst_ptr + offset_dst,
                            vector_ptr + offset_vector,
                            (vector_depth) * sizeof(*vector_ptr));
            }

        }, src_iter);

}
```

GPU vectorize kernel (Listing 3.5) chose a different approach to populate output tensor, it configures window to iterate every element in output tensor in shape ($\mathcal{L}$, d_{model}). Then uses configured window to launch in total $\mathcal{L}$ * d_{model} threads, each thread populates one scalar in output tensor. The kernel retrieves global thread IDs (id_x, id_y, id_z), which correspond to the coordinates within tensors. These IDs are used to compute the linearized indices for accessing data within the tensors.

Listing 3.5: GPU Vectorize Kernel

```
__kernel void vectorize(TENSOR3D_DECLARATION(src),
                        TENSOR3D_DECLARATION(vector),
                        TENSOR3D_DECLARATION(output))
{
    int id_x = get_global_id(0);
    int id_y = get_global_id(1);
    int id_z = get_global_id(2);

    int out_linear_idx = id_y * output_stride_y + id_x * output_stride_x;
    int src_linear_idx = id_y * src_stride_x;

    src_ptr += src_offset_first_element_in_bytes + src_linear_idx;

    int vector_offset = *((__global int *)src_ptr);
    int vector_linear_idx = vector_offset * vector_stride_y + id_x *
        vector_stride_x;

    output_ptr += output_offset_first_element_in_bytes + out_linear_idx;
    vector_ptr += vector_offset_first_element_in_bytes + vector_linear_idx;
    *((__global DATA_TYPE_DST *)output_ptr) = *((__global DATA_TYPE_VEC *)
        vector_ptr);
}
```

3.3. Multi-Head Attention

MHA layer contains two sub-layers, MHA linear layer and SDPA layer.

3.3.1. Multi-Head Attention Linear

MHA linear layer projects input with pre-trained parameter to yield Query, Key and Value. It contains three computationally intensive linear operations, which perform matrix multiplication (MMUL) and transform the input data:

$$y = \alpha \cdot xA^T + B$$

Where y, x denotes output and input matrix respectively, A and B are pre-trained weights and biases in matrix or vector. α is the scaling factor for product between input and weight.

CPU and GPU linear operation has distinctive approach to utilize hardware acceleration. To use NEON vectorization in CPU implementation, left-hand side matrix is first performed 4*4 interleave transformation, where data in a 4*4 tensor block is put in the same row. As in later MMUL computation, data needs to be loaded into vector lane column-wise. Right-hand side matrix is performed 1*W transformation, where tensor rows are sequentially connected to output a tensor with shape $(1, \mathcal{L} * d_{model})$. Such pre-process enabled efficient memory access and vectorization in following MMUL. If transposition is needed, right-hand side matrix is first transposed before 1*W transformation.

The GPU linear operation performs tile MMUL, with each GPU thread assigned to compute a block of the MMUL product in the output tensor, as shown in Figure 3.4. Unlike the CPU linear operation, the right-hand side transposition can be performed in place during

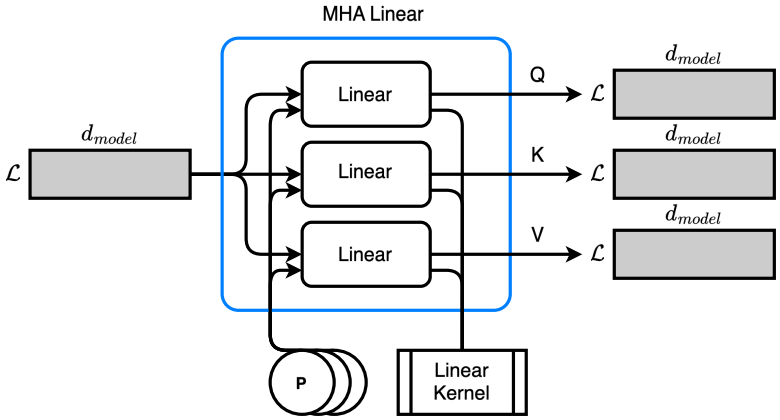

Figure 3.2: MHA linear layer contains three linear operations, loads three pre-trained model parameters and produces Query, Key, Value tensors.

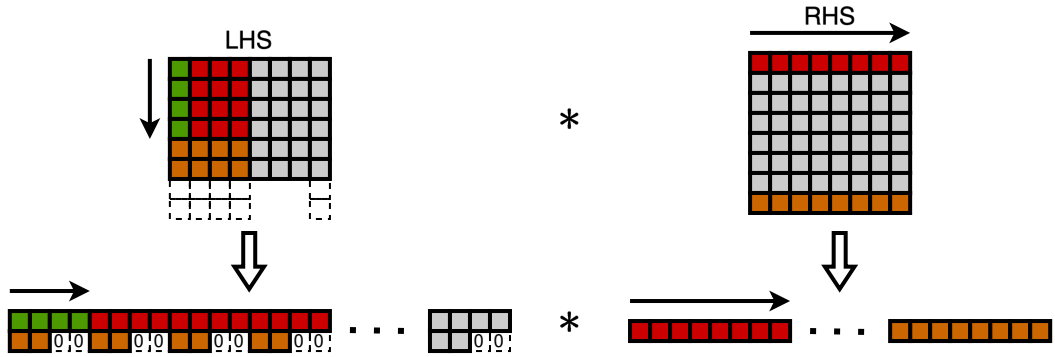

Figure 3.3: CPU linear pre-process, one cell represents one scalar data within tensor. Left: 4*4 interleave transformation, Right: 1*W transformation. LHS 4*4 interleave boundary 4*4 block is padded with 0.

computation. The GPU kernel can be configured to adjust the memory access pattern either row-wise or column-wise, enabling RHS transposed or non-transposed MMUL.

3.3.2. Linear Kernel

Listing 3.6 shows the CPU linear operation, which wraps over 4*4 interleave, 1*W transformation and MMUL. Each operation takes a tensor pack which contains tensors from previous connected nodes, and initializes auxiliary tensor handlers if intermediate memory space is needed for computing. Then schedules necessary kernel computation.

Listing 3.6: CPU Linear Operation

```
1 void CpuLinear::run(ITensorPack &tensors)
2 {
3     auto a = tensors.get_const_tensor(ACL_SRC_0);
4     auto b = tensors.get_const_tensor(ACL_SRC_1);
5     auto c = tensors.get_const_tensor(ACL_SRC_2);
6     auto d = tensors.get_tensor(ACL_DST);
7
```

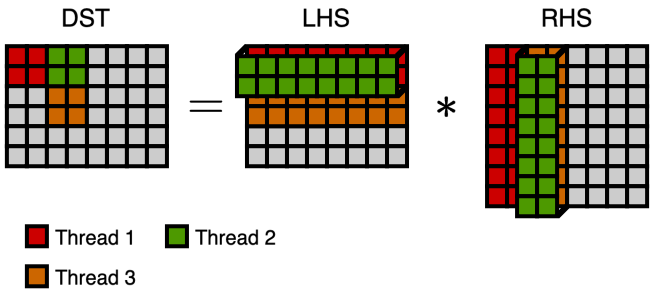

Figure 3.4: GPU RHS non-transposed tile MMUL example. Thread 1 is assigned to compute the output tensor data from (0,0) to (1,1). It accesses rows 0 and 1 in the left-hand side matrix and columns 0 and 1 in the right-hand side matrix to compute the MMUL product.

```
8     CpuAuxTensorHandler interleaved_a(InterleavedLHS, _tmp_a);
9     CpuAuxTensorHandler pretransposed_b(PreTransposedRHS, _pretransposed_b);
10    CpuAuxTensorHandler transposed1xw_b(Transposed1xWRHS, _tmp_b);
11    CpuAuxTensorHandler temp_d(TempResult, _tmp_d);
12
13    ITensorPack mm_pack{ { ACL_SRC_0, a },
14                         { ACL_SRC_1, b },
15                         { ACL_DST, (_run_bias_addition) ? temp_d.get() : d } };
16
17    if(_run_interleave_transpose)
18    {
19        ITensorPack interleave_pack{ { ACL_SRC, a },
20                                     { ACL_DST, interleaved_a.get() } };
21        NEScheduler::get().schedule_op(_interleave_kernel.get(),
22                                       _interleave_kernel->window(),
23                                       interleave_pack);
24        mm_pack.add_const_tensor(ACL_SRC_0, interleaved_a.get());
25    }
26
27    const ITensor *b_to_use = b;
28
29    if(_pretranspose_b_func)
30    {
31        ITensorPack pretranspose_pack{ { ACL_SRC, b_to_use },
32                                       { ACL_DST, pretransposed_b.get() } };
33        _pretranspose_b_func->run(pretranspose_pack);
34        b_to_use = pretransposed_b.get();
35    }
36
37    if(_run_interleave_transpose)
38    {
39        ITensorPack transpose_pack{ { ACL_SRC, b_to_use },
40                ‘                   { ACL_DST, transposed1xw_b.get() } };
41        NEScheduler::get().schedule_op(_transpose1xW_b_kernel.get(),
42                                       _transpose1xW_b_kernel->window(),
43                                       transpose_pack);
44
45        b_to_use = transposed1xw_b.get();
46    }
47
48    mm_pack.add_const_tensor(ACL_SRC_1, b_to_use);
```

```
49
50      NEScheduler::get().schedule_op(_mm_kernel.get(),
51                                     _mm_kernel->window(),
52                                     mm_pack);
53
54      if(_run_bias_addition)
55      {
56          ITensorPack pack{ { ACL_SRC_0, temp_d.get() },
57                            { ACL_SRC_1, c },
58                            { ACL_DST, d } };
59          NEScheduler::get().schedule_op(_add_bias.get(),
60                                         _add_bias->window(),
61                                         pack);
62      }
```

GPU linear operation schedules only one kernel, but calls different kernel implementations depending on transposing LHS or RHS tensor. Listing 3.7 shows the LHS non-transpose, RHS transpose example, which accesses RHS tensor memory row-wise for transposed computation. GPU implementation uses Marcos to increase inline performance and code readability. As in some cases GPU tile MMUL has unevenly divided row numbers, GPU need to launch extra threads. To save branch scheduling costs, some threads will compute on same data by using Marco 'GET_SPATIAL_IDX'.

The kernel uses 'TILE' to initialize accumulator (acc) memory region to store intermediate results for each tile of the matrix multiplication, and a temporary buffer (ret) to hold the final results for each thread's portion of the output. 'TILE' memory allocation and access wraps over OpenCL vector data type such as 'double*n*' to utilize OpenCL vector computation. Then unroll loop using 'LOOP_UNROLLING' to perform multiplication, if number of columns (K) is not evenly divisible by the 'TILE' width (K0), left-over computation will be performed.

The multiplication result is then bias-added, scaled by constant 'Alpha' and 'Beta' if such computation is configured. As some threads may be duplicate computing on the same data, 'indirect_buffer' and 'T_STORE_INDIRECT_WIDTH_SELECT' are used to perform selective storing and ensure correct results in edge cases.

Listing 3.7: GPU linear kernel LHS non-transpose, RHS transpose

```
1  __kernel void mat_mul_mmul_nt_t(
2      TENSOR3D_T(lhs, BUFFER),
3      TENSOR3D_T(rhs, RHS_TENSOR_TYPE),
4  #ifdef BIAS
5      TENSOR3D_T(bias, BUFFER),
6  #endif // defined(BIAS)
7      TENSOR3D_T(dst, BUFFER))
8  {
9      uint x = GET_SPATIAL_IDX(0, N0, PARTIAL_STORE_N0);
10     uint y = GET_SPATIAL_IDX(1, M0, PARTIAL_STORE_M0);
11     uint z = GET_SPATIAL_IDX(2, 1, 0);
12
13     lhs_offset_first_element_in_bytes += y * lhs_stride_y + z * lhs_stride_z;
14     dst_offset_first_element_in_bytes += x * sizeof(DATA_TYPE) + y * dst_stride_y
           +
15                                          z * dst_stride_z;
16
```

```
    TILE(DATA_TYPE, M0, N0, ret);
    TILE(DATA_TYPE, M0, N0, acc);
    T_LOAD_TILE(DATA_TYPE, M0, N0, BUFFER, lhs, 0, 0, 1, lhs_stride_y, acc);

    LOOP_UNROLLING(int, _m, 0, 1, M0,
    {
        LOOP_UNROLLING(int, _n, 0, 1, N0,
        {
            TILE_ACCESS(acc,_m,_n,N0) = 0.f;
        })
    })

    const int rhs_z = z * rhs_h;
    int       k;
    for(k = 0; k <= K - K0; k += K0)
    {
        TILE(DATA_TYPE, M0, K0, a);
        TILE(DATA_TYPE, N0, K0, b);

        T_LOAD_TILE(DATA_TYPE, M0, K0, BUFFER, lhs, k,
                                            0, 1, lhs_stride_y, a);
        T_LOAD_TILE(DATA_TYPE, N0, K0, BUFFER, rhs, k,
                                    x + rhs_z, 1, rhs_stride_y, b);

        LOOP_UNROLLING(int, _m, 0, 1, M0,
        {
            LOOP_UNROLLING(int, _n, 0, 1, N0,
            {
                LOOP_UNROLLING(int, _k, 0, 1, K0,
                {
                    TILE_ACCESS(acc, _m, _n ,N0) =
                    fma((DATA_TYPE)TILE_ACCESS(a, _m, _k, K0),
                        (DATA_TYPE)TILE_ACCESS(b, _n, _k, K0),
                        (DATA_TYPE)TILE_ACCESS(acc, _m, _n, N0));
                })
            })
        })

    }

#if K % K0 != 0
    /* Compute leftover Loop */
    ...
    /* Compute leftover Loop */
#endif // K % K0 != 0

    const bool x_cond = PARTIAL_STORE_N0 != 0 && get_global_id(0) == 0;
    const bool y_cond = PARTIAL_STORE_M0 != 0 && get_global_id(1) == 0;

    TILE(int, M0, 1, indirect_buffer);
    LOOP_UNROLLING(int, _i, 0, 1, M0,
    {
        indirect_buffer[_i].v = min(_i,
                                    select(M0 - 1, PARTIAL_STORE_M0 - 1, y_cond))
                                        ;
    });
```

```
72
73 #ifdef BIAS
74     TILE(DATA_TYPE, 1, N0, bias_tile);
75     T_LOAD_TILE(DATA_TYPE, 1, N0, BUFFER, bias, x, 0, 1, 0, bias_tile);
76
77     LOOP_UNROLLING(int, _m, 0, 1, M0,
78     {
79         LOOP_UNROLLING(int, _n, 0, 1, N0,
80         {
81             TILE_ACCESS(acc,_m,_n,N0) += bias_tile[_n];
82         })
83     })
84 #endif // defined(BIAS)
85
86     LOOP_UNROLLING(int, _m, 0, 1, M0,
87     {
88         LOOP_UNROLLING(int, _n, 0, 1, N0,
89         {
90             TILE_ACCESS(acc,_m,_n,N0) = TILE_ACCESS(acc,_m,_n,N0)
91                                         * ALPHA + BETA;
92         })
93     })
94
95     for(int _m = 0; _m < M0; _m++)
96     {
97         ret[_m].v = V_LOAD_TILE(DATA_TYPE, N0, acc, 0, _m, N0);
98     }
99
100    T_STORE_INDIRECT_WIDTH_SELECT(DATA_TYPE, M0, N0, PARTIAL_STORE_N0,
101                                  BUFFER, dst, 0, dst_stride_y, x_cond,
102                                  ret, indirect_buffer);
103
104 }
```

3.3.3. Scale Dot Production Attention

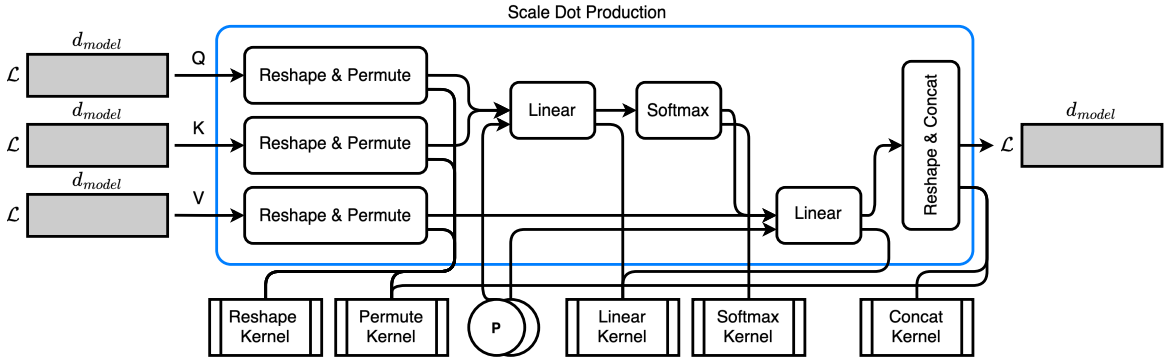

Figure 3.5: Scale dot production attention workflow.

SDPA is designed with h parallel heads, where the attention score is computed independently by each head. Each head tensor has a reduced dimension from d_{model} to $d_{head} = \frac{d_{model}}{h}$. In the BERT base uncased model, $d_{head} = \frac{d_{model}}{h} = \frac{768}{12} = 64$. To produce parallel heads, input tensors

are reshaped and permuted to utilize the batch dimension, transforming from a 2D shape ($\mathcal{L}$, d_{model}) into a 3D shape ($\mathcal{L}$, d_{head}, h). The batched tensors are then used to compute the scaled dot production.

Query and Key tensors are multiplied using MMUL and scaled by $\alpha = \frac{1}{\sqrt{d_k}}$ to produce the scaled product using the linear kernel. The MMUL product is computed between corresponding batches as parallel heads. The scaled product is then applied with softmax and multiplied with the Value tensor. Finally, the result is reshaped and concatenated to produce SDPA output.

Listing 3.8 shows the GPU SDPA operation. Query, Key, and Value input are first reshaped and permuted to match the required dimensions for multi-head attention. Then perform matrix multiplication between Query and Key attention heads to calculate scaled dot-product, result is softmaxed and multiplied with Value. The computed result from all attention heads is concatenated together and then permuted to restore the original ordering of dimensions. CPU SDPA operation shares the same structure as GPU implementation.

Listing 3.8: GPU SDPA operation 'run' function implementation

```
void ClScaleDotProduction::run(ITensorPack &tensors)
{
    auto query  = tensors.get_const_tensor(ACL_SRC_0);
    auto key    = tensors.get_const_tensor(ACL_SRC_1);
    auto value  = tensors.get_const_tensor(ACL_SRC_2);
    auto output = tensors.get_tensor(ACL_DST);

    CLAuxTensorHandler reshaped_query(QueryReshape, _reshaped_query);
    /* More auxiliary memory handles */
        ...
    /* More auxiliary memory handles */
    CLAuxTensorHandler permuted_concat(ConcatPermute, _permuted_concat);

    // Run Query multi-Head reshape, permute
    ITensorPack query_reshape_pack{ { ACL_SRC_0, query },
                                    { ACL_DST, reshaped_query.get() } };
    CLScheduler::get().enqueue_op(*_query_reshape_kernel, query_reshape_pack);
    ITensorPack query_permute_pack{ { ACL_SRC, reshaped_query.get() },
                                    { ACL_DST, permuted_query.get() } };
    CLScheduler::get().enqueue_op(*_query_permute_kernel, query_permute_pack);

    /* Key and Value multi-Head reshape, permute */
        ...
    /* Key and Value multi-Head reshape, permute */

    // Run matrix multiply compute multi-head attention between Query and Key
    ITensorPack linear_QK_pack{ { ACL_SRC_0, permuted_query.get() },
                                { ACL_SRC_1, permuted_key.get() },
                                { ACL_DST, scaled_query_key.get() } };
    CLScheduler::get().enqueue_op(*_product_mm_kernel, linear_QK_pack);

    // Softmax scaled product
    ITensorPack softmax_pack = { { ACL_SRC, scaled_query_key.get() },
                                 { ACL_DST, softmaxed_product.get() } };
    CLScheduler::get().enqueue_op(*_softmax_kernel, softmax_pack);

```

```
37      // Run matrix multiply compute multi-head attention between Context and Value
38      ITensorPack linear_context_pack{ { ACL_SRC_0, softmaxed_product.get() },
39                                       { ACL_SRC_1, permuted_value.get() },
40                                       { ACL_DST, lineared_context.get() } };
41      CLScheduler::get().enqueue_op(*_context_mm_kernel, linear_context_pack);
42
43      // Concat all attention head together
44      ITensorPack concat_permute_pack{ { ACL_SRC, lineared_context.get() },
45                                       { ACL_DST, permuted_concat.get() } };
46      CLScheduler::get().enqueue_op(*_concat_permute_kernel,concat_permute_pack);
47      ITensorPack concat_reshape_pack{ { ACL_SRC_0, permuted_concat.get() },
48                                       { ACL_DST, output } };
49      CLScheduler::get().enqueue_op(*_concat_reshape_kernel, concat_reshape_pack);
50  }
```

3.4. Add&Norm, Feed Forward

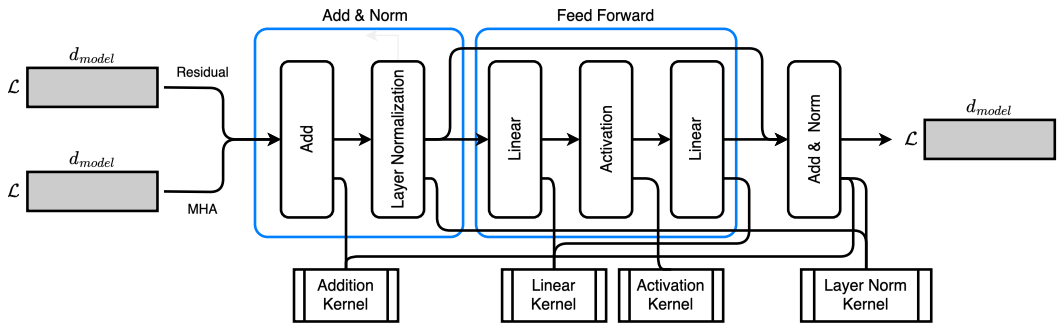

Figure 3.6: Add&Norm and Feed Forward layer.

3.4.1. Add&Norm

The MHA output tensor is element-wise added to the residual connection tensor from the MHA input. The resulting tensor is then subjected to layer normalization. Transformer models adopt layer normalization instead of batch normalization [36]. Layer normalization is calculated within each feature vector l:

$$\mu^l = \frac{\sum_{i=1}^{d_{model}} x_i^l}{d_{model}}$$

$$\sigma^l = \sqrt{\frac{\sum_{i=1}^{d_{model}} (x_i^l - \mu^l)^2}{d_{model}}}$$

Where x_i^l is i_{th} scalar value within feature vector l of length d_{model}. μ^l denotes layer mean and σ^l denotes layer variance.

3.4.2. Layer Normalization Kernel

Listing 3.9 depicts GPU layer normalization kernel. Each thread is assigned to calculate one feature vector, which is in the x dimension of each tensor. It computes vector mean

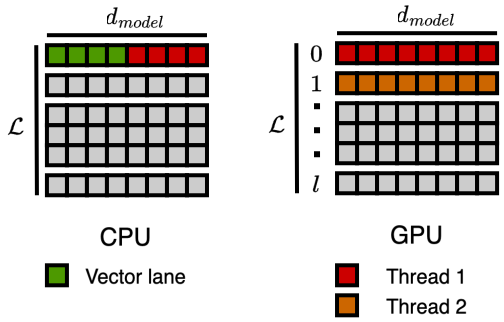

Figure 3.7: CPU layer normalization uses vectorization to accumulate result, GPU layer normalization launches total L threads to calculate each feature.

and variance, then normalized scalar element value. It also uses vectorized operations to process multiple elements in parallel, VLOAD, VSTORE and SUM are Marcos that expands to corresponding OpenCL vector data load, store and reduction respectively. CPU layer normalization shares the same structure, except using NEON equivalent vector instruction.

Listing 3.9: GPU layer normalization kernel

```
__kernel void layer_norm(TENSOR3D_DECLARATION(input),
                         TENSOR3D_DECLARATION(output),
                         DATA_TYPE epsilon,
                         DATA_TYPE gamma,
                         DATA_TYPE beta)
{
    int y = get_global_id(1);
    int z = get_global_id(2);

    __global uchar *input_addr  = input_ptr +
                                  input_offset_first_element_in_bytes +
                                  y * input_stride_y;
    __global uchar *output_addr = output_ptr +
                                  output_offset_first_element_in_bytes +
                                  y * output_stride_y;

    DATA_TYPE res = (DATA_TYPE)0;
    DATA_TYPE mean;
    DATA_TYPE var = (DATA_TYPE)0;
    DATA_TYPE sqrt_var_epsilon;

    // Calculate mean
    int x = 0;
    for(; x <= (WIDTH - VEC_SIZE); x += VEC_SIZE)
    {
        VEC_DATA_TYPE(DATA_TYPE, VEC_SIZE) vals
            = VLOAD(VEC_SIZE)(0, (__global DATA_TYPE *)(input_addr +
                                                        x * input_stride_x));
        res  = SUM(res, vals, VEC_SIZE);
    }

// Mean left-over
#if(WIDTH % VEC_SIZE)
```

```
    for(; x < WIDTH; ++x)
    {
        DATA_TYPE val = *((__global DATA_TYPE *)(input_addr +
                                                 x * input_stride_x));
        res           = SUM(res, val, 1);
    }

#endif // (WIDTH % VEC_SIZE)

    mean = res / WIDTH;
    VEC_DATA_TYPE(DATA_TYPE, VEC_SIZE) means = mean;

    // Calculate variance
    x = 0;
    for(; x <= (WIDTH - VEC_SIZE); x += VEC_SIZE)
    {
        VEC_DATA_TYPE(DATA_TYPE, VEC_SIZE) vals
            = VLOAD(VEC_SIZE)(0, (__global DATA_TYPE *)(input_addr +
                                                        x * sizeof(DATA_TYPE)));
        vals  = vals - mean;
        vals  = vals * vals;
        var   = SUM(var, vals, VEC_SIZE);
    }

// Variance left-over
#if(WIDTH % VEC_SIZE)
    for(; x < WIDTH; ++x)
    {
        DATA_TYPE val = *((__global DATA_TYPE *)(input_addr +
                                                 x * sizeof(DATA_TYPE)));
        val           = val - mean;
        val           = val * val;
        var           = SUM(var, val, 1);
    }
#endif // (WIDTH % VEC_SIZE)

    var = var / WIDTH;
    sqrt_var_epsilon = sqrt(var + epsilon);
    VEC_DATA_TYPE(DATA_TYPE, VEC_SIZE) sqrt_var_epsilons = sqrt_var_epsilon;
    VEC_DATA_TYPE(DATA_TYPE, VEC_SIZE) gammas = gamma;
    VEC_DATA_TYPE(DATA_TYPE, VEC_SIZE) betas = beta;

    // Calculate layer normalization
    x = 0;
    for(; x <= (WIDTH - VEC_SIZE); x += VEC_SIZE)
    {
        VEC_DATA_TYPE(DATA_TYPE, VEC_SIZE) vals
            = VLOAD(VEC_SIZE)(0, (__global DATA_TYPE *)(input_addr +
                                                        x * sizeof(DATA_TYPE)));
        vals = vals - mean;
        vals = vals / sqrt_var_epsilons;
        vals = vals * gammas;
        vals = vals + betas;

        VSTORE(VEC_SIZE)(vals, 0, (__global DATA_TYPE *)(output_addr +
                                                         x * output_stride_x));
```

```
90      }
91
92  // Layer normalization left-over
93  #if(WIDTH % VEC_SIZE)
94      for(; x < WIDTH; ++x)
95      {
96          DATA_TYPE val = *((__global DATA_TYPE *)(input_addr + x * sizeof(
                DATA_TYPE)));
97          val = val - mean;
98          val = val / sqrt_var_epsilon;
99          val = val * gamma;
100         val = val + beta;
101         VSTORE(1)(val, 0, (__global DATA_TYPE *)(output_addr + x *
                output_stride_x));
102
103     }
104 #endif // (WIDTH % VEC_SIZE)
105 }
```

3.4.3. Feed Forward

Transformer employed a feed-forward mechanism which consists of 2 linear operations and an activation operation in between, as defined in the model graph description within encoder block as depicted in Listing 3.2. Output from previous layer normalization is projected from d_{model} to d_{ff} through linear kernel MMUL with pre-trained parameters, where $d_{ff} = 3072$ for BERT base uncased model. Projected tensor is activated through GELU function, and projects back to d_{model} after the consecutive linear operation.

4

Evaluation & Experimental Results

4.1. Experimental Setup

This work used Khadas Vim 3, which is equipped with Amlogic A311D HMPSoC. This device has a ARM big.Little architecture multi-core CPU and a Mali G52 GPU. CPU contains four Cortex-A73 cores for the big cluster and 2 Cortex-A53 cores for the Little cluster. Maximum frequency for big cluster is 1.8GHz and 0.8GHz for G52 GPU. Device uses 2GB LPDDR4 as main memory.

Device was running Ubuntu 22.04.3 LTS (Jammy Jellyfish), and this work developed Transformer inference based on ARM-CL v24.01. Experiment used data type FP32 for both CPU and GPU, CPU result is based on 4 threads running on big cluster. All experiments do not use extra tuner and are repeated 10 times to get mean result.

4.2. Layer Latency Analysis

The Transformer model is distinct from CNNs and has four significant layers: the MHA Linear layer, the SDPA layer, the Layer Normalization layer, and the FF layer. The MHA Linear layer and FF layer are computationally intensive, primarily performing MMUL operations. The SDPA layer is the most complex due to its extra memory-intensive reshape and permute operations. This section will analyze the characteristics of each layer running on different HMPSoC components.

Figure 4.1 shows the mean inference latency of each layer for BERT base uncased model. The attention linear layer and the FF layer account for most of the inference latency in the CPU implementation, while the SDPA layer takes the most inference time on the GPU. This paper defines $T_{CPU/GPU} = \frac{t_{CPU}}{t_{GPU}}$ to denote the layer latency ratio between CPU and GPU inference as shown in Table 4.1. The results suggest that layers involving computationally intensive matrix multiplication with linear kernels perform better on the GPU than on the CPU. This paper further explored performance across different parameter spaces to analyze layer runtime characteristics. Since the computational workload is based on tensors and most operations involve tensors shaped ($\mathcal{L}$,d_{model}), This paper first examined the processed token length $\mathcal{L}$ and model depth d_{model}.

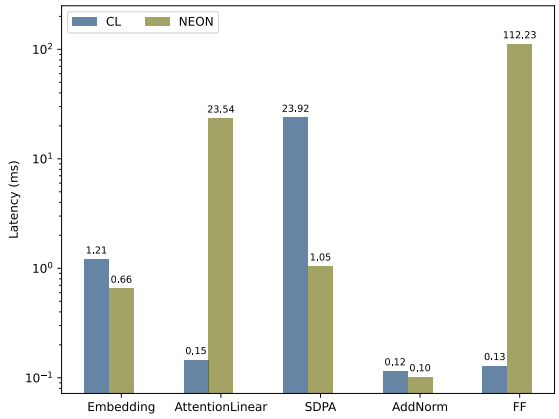

Figure 4.1: Mean layer inference latency comparison for BERT base uncased between CPU and GPU. $\mathcal{L} = 16$, $d_{model} = 768$.

Layers	$T_{CPU/GPU}$
Embedding	0.54
Attention Linear	160.90
SDPA	0.04
Layer Norm	0.88
FF	869.32

Table 4.1: CPU and GPU layer inference latency ratio $T_{CPU/GPU}$ from Figure 4.1 .

Figure 4.2 compares the efficiency of different layers running on CPU and GPU. We observed that the **attention linear layer and FF layer perform significantly better on the GPU than on the CPU** across all token lengths $\mathcal{L}$ and model depth d_{model}. Embedding layer and add&norm layer show comparable performance on both CPU and GPU, although the GPU performs better as the tensor shape increases with larger $\mathcal{L}$ and d_{model}. While **SDPA layer consistently performs better on the CPU**.

4.2.1. Multi-Head Attention Linear

MHA linear layer involves three linear kernel computations on tensors of the same shape. Input left-hand side tensor has shape ($\mathcal{L}$,d_{model}) and right-hand side tensor has shape (d_{model},d_{model}). Total arithmetic computation for one MMUL is $\mathcal{L} * d_{model} * d_{model}$, which has $O(n^3)$ complexity, while memory access is $\mathcal{L} * d_{model} + d_{model} * d_{model}$ which has $O(n^2)$ complexity. GPU linear kernel employed tile MMUL, distributing computation across multiple cores. Each GPU core's computing complexity has a smaller base $O(n)$ compared to CPU linear kernel. For $d_{model} = 768$, each GPU thread computes $6 * 2 * 768$ arithmetic operation. This approach utilize multiple cores to benefit computation-intensive tasks by improving total throughput, as $O(n^3)$ complexity scales less dramatically when d_{model} increases. Which also explains GPU became more efficient computing such layer when d_{model} increases.

4.2.2. Feed Forward

GPU also shows dominant efficiency in computing the FF layer compared to the CPU, and FF layer accounts for the majority of the computation time in CPU inference. FF layer includes two MMUL linear kernel computations and an element-wise GELU activation in between 2 linear layers. FF layer uses MMUL to project tensor into depth of $d_{ff} = 3072$, with $\mathcal{L} * d_{model} * d_{ff}$ arithmetic computation, making the FF layer even more computation-intensive compared to the attention linear layer.

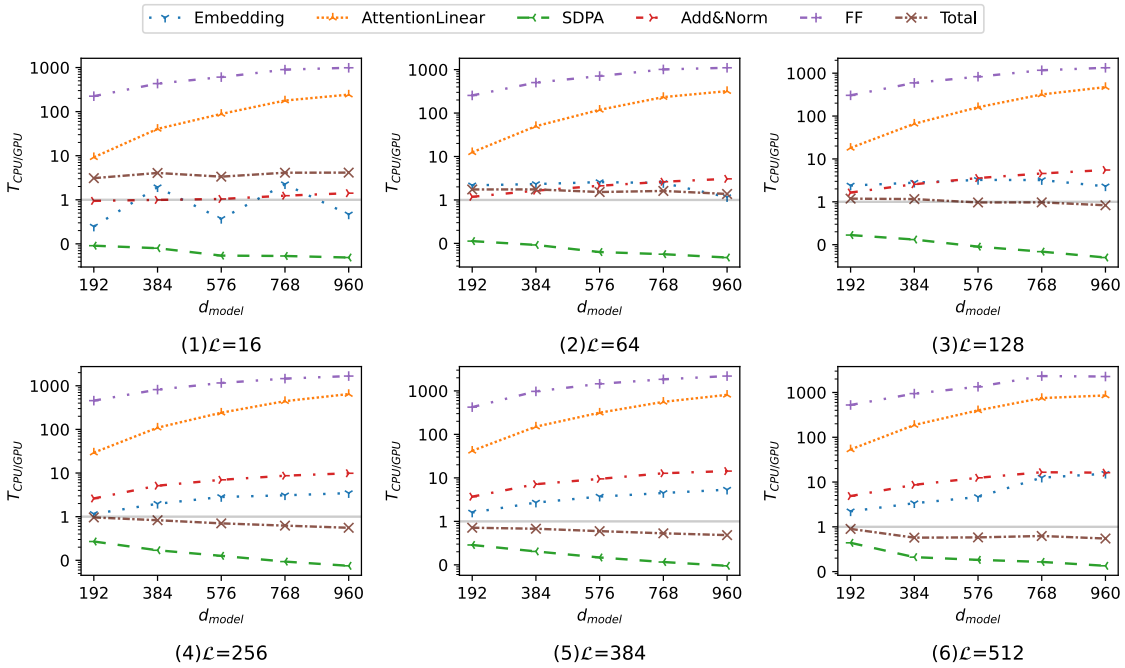

Figure 4.2: Exploring $T_{CPU/GPU}$ of model depth d_{model} with different token length $\mathcal{L}$. y-axis in log scale

4.2.3. Embedding, Add&Norm

The embedding layer and the add & norm layer exhibit comparable efficiency on both CPU and GPU, but the CPU demonstrates overall better efficiency as vector depth and token length increase. This is attributed to the CPU's lower memory access latency compared to the GPU, as these layers primarily involve memory operations. Embedding layer consists of three vectorize kernels, each accessing $\mathcal{L} * d_{model}$ memory. This is followed by an element-wise addition kernel that accesses the previously produced three $\mathcal{L} * d_{model}$ memory and performs $2 * \mathcal{L} * d_{model}$ arithmetic computations.

The add&norm layer first accesses two $\mathcal{L} * d_{model}$ tensors and performs $\mathcal{L} * d_{model}$ arithmetic additions to aggregate the residual connection. Layer normalization then accesses the $\mathcal{L} * d_{model}$ tensor memory three times to compute the mean, variance, and normalized product. Since the tensor memory within the same feature vector l is row-adjacent, the CPU layer normalization kernel can easily apply NEON vectorization. In contrast, the GPU implementation requires each work thread to sequentially access the feature vector of d_{model} length, leading to less efficient memory access patterns.

4.3. Design Space Exploration

4.3.1. Scale Dot Production Attention Analysis

The CPU consistently demonstrates better efficiency in computing the SDPA layer compared to the GPU. Additionally, we observed that SDPA accounts for nearly all of the computing time in GPU inference. We define $T_{SDPA} = \frac{t_{SDPA}}{\sum t_{layers}}$ to represent the proportion of SDPA computing

time relative to the total layer latency. Table 4.2 presents the comparison of T_{SDPA} between GPU and CPU across different workloads.

Vector Depth	192		384		576		768		960	
$\mathcal{L}$	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU
16	0.925	0.027	0.951	0.018	0.972	0.015	0.977	0.012	0.980	0.011
64	0.966	0.062	0.983	0.051	0.990	0.041	0.992	0.035	0.994	0.034
128	0.982	0.139	0.991	0.113	0.995	0.093	0.996	0.071	0.980	0.060
256	0.992	0.280	0.996	0.205	0.997	0.181	0.998	0.151	0.998	0.135
384	0.994	0.403	0.997	0.302	0.998	0.249	0.999	0.220	0.999	0.198
512	0.995	0.490	0.998	0.369	0.998	0.316	0.999	0.264	0.999	0.246

Table 4.2: Comparison of SDPA latency percentages to total layer inference latency between CPU and GPU.

Original paper claimed parallel attention head reduced theoretical linear computation dimension for each head, the total computational cost is similar to that of single-head attention with full dimensionality. But for edge inference, this comes at the cost of adding extra memory operation and runtime scheduling. So this paper further explored the impact of attention heads on SDPA layer.

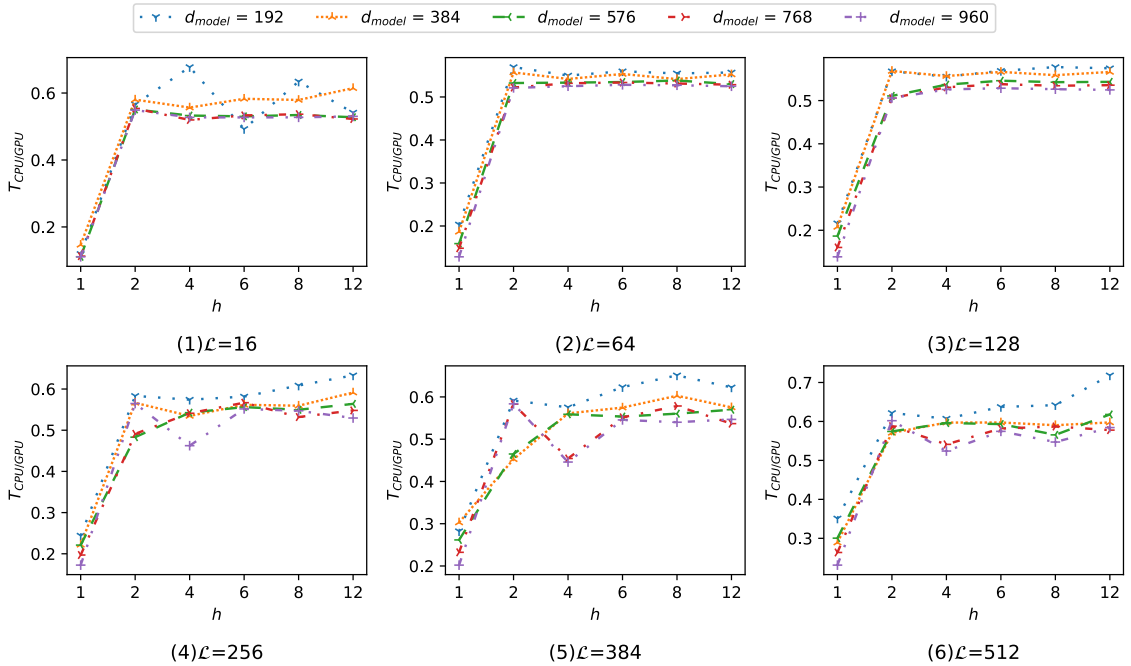

Figure 4.3: Exploring $T_{CPU/GPU}$ of attention heads h with different model depth d_{model} and token length $\mathcal{L}$.

Figure 4.3 illustrates the SDPA inference efficiency $T_{CPU/GPU}$ with varying attention heads h. The CPU significantly outperforms the GPU when $h = 1$, and maintains approximately

double the efficiency with two or more heads. This discrepancy arises because the inference latency for the GPU is minimally impacted by the number of attention heads, though this comes at the expense of an overall longer inference time. Conversely, the CPU's latency increases with the number of attention heads, showing a notable jump from $h = 1$ to $h = 2$. Detailed runtime data is provided in Appendix Table A.1.

4.3.2. Feed Forward Layer Depth Analysis

Given that the FF layer constitutes two-thirds of a transformer model's parameters [11], and significantly impacts CPU inference latency, this paper investigated the effect of d_{ff}. Figure 4.4 presents the experimental results, demonstrating that the GPU exhibits superior efficiency in computing the FF layer compared to the CPU across all workloads. The GPU's inference latency remains relatively unaffected by d_{ff}, except when $d_{model} = 960$. In contrast, the CPU's latency scales with d_{ff}. Detailed runtime data is available in Appendix Table A.2.

We observed a decrease in GPU efficiency as the workload increases for the case where $d_{model} = 960$ and $d_{ff} \geq 3072$ case. This paper attribute this to GPU memory access and thread scheduling issues. Despite this, the GPU maintains better efficiency compared to CPU.

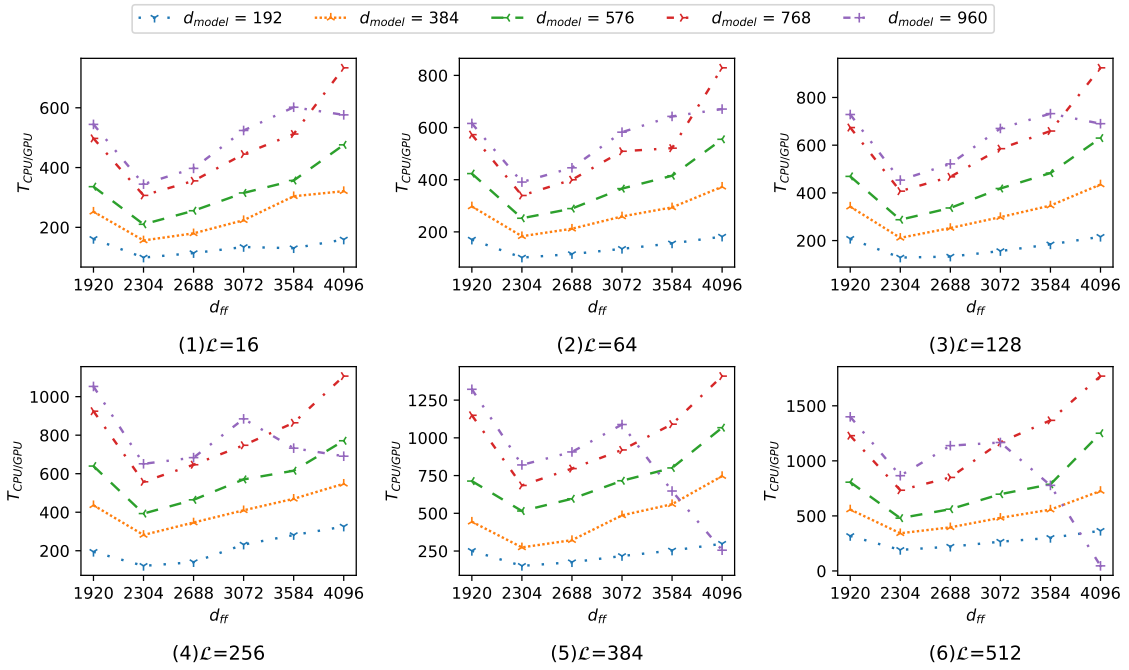

Figure 4.4: Exploring $T_{CPU/GPU}$ of feed forward layer depth d_{ff} with different model depth d_{model} and token length $\mathcal{L}$.

5

Conclusion

This work implemented Transformer edge inference using the ARM-CL, achieving a significant reduction in inference latency compared to existing methods. And conducted a comprehensive analysis of layer characteristics and performed several experiments to explore layer performance on both CPU and GPU under varying workloads.

Our observations revealed that the Transformer model exhibits distinctive layer runtime performance. The GPU demonstrates high efficiency in computing attention linear layers and FF layers, which involve intensive MMUL and maintains overall stable inference latency across different workloads. In contrast, while the CPU's performance scales with the workload, it is particularly efficient at computing the SDPA layer.

For future work, we aim to implement a CPU-GPU layer-switching approach instead of relying solely on CPU or GPU inference. We anticipate that layer-switching inference can further reduce overall inference latency, given the significant differences in runtime latency for various layers running on CPU and GPU. We hope these findings will enhance our understanding of Transformer edge inference and inspire the design of more efficient models, particularly in the context of model compression and pruning.

Acknowledgement

I would like to thank my mother for laying the foundation that has allowed me to pursue myself.

I also extend my gratitude to Amsterdam and everyone along the way, helped my through sun and rain, sun again, rain again.

References

[1] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[2] W. Liu, D. Anguelov, D. Erhan, *et al.*, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, Springer, 2016, pp. 21–37.

[3] C. Yu, J. Wang, C. Peng, C. Gao, G. Yu, and N. Sang, "Bisenet: Bilateral segmentation network for real-time semantic segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 325–341.

[4] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[5] P. Marwedel, *Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things*. Springer Nature, 2021.

[6] M. Abadi, A. Agarwal, P. Barham, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[7] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[8] A. Ltd. "A software library for machine learning." (2023), [Online]. Available: `https://www.arm.com/technologies/compute-library` (visited on 06/03/2023).

[9] T. Chen, T. Moreau, Z. Jiang, *et al.*, "{Tvm}: An automated {end-to-end} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[10] O. Press, N. A. Smith, and O. Levy, "Improving transformer models by reordering their sublayers," *arXiv preprint arXiv:1911.03864*, 2019.

[11] M. Geva, R. Schuster, J. Berant, and O. Levy, "Transformer feed-forward layers are key-value memories," *arXiv preprint arXiv:2012.14913*, 2020.

[12] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training," 2018.

[13] S. Kim, C. Hooper, T. Wattanawong, *et al.*, "Full stack optimization of transformer inference: A survey," *arXiv preprint arXiv:2302.14017*, 2023.

[14] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, 2017.

[15] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "Mobilebert: A compact task-agnostic bert for resource-limited devices," *arXiv preprint arXiv:2004.02984*, 2020.

[16] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[17] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.

[18] F. N. Iandola, A. E. Shaw, R. Krishna, and K. W. Keutzer, "Squeezebert: What can computer vision teach nlp about efficient neural networks?" *arXiv preprint arXiv:2006.11316*, 2020.

[19] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[20] P. Michel, O. Levy, and G. Neubig, "Are sixteen heads really better than one?" *Advances in neural information processing systems*, vol. 32, 2019.

[21] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[22] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[23] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.

[24] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[25] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[26] D. Chen, Y. Li, M. Qiu, *et al.*, "Adabert: Task-adaptive bert compression with differentiable neural architecture search," *arXiv preprint arXiv:2001.04246*, 2020.

[27] J. Xu, X. Tan, R. Luo, *et al.*, "Nas-bert: Task-agnostic and adaptive-size bert compression with neural architecture search," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 1933–1943.

[28] R. Wang, Q. Bai, J. Ao, *et al.*, "Lighthubert: Lightweight and configurable speech representation learning with once-for-all hidden-unit bert," *arXiv preprint arXiv:2203.15610*, 2022.

[29] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "μLayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[30] E. Aghapour, D. Sapra, A. Pimentel, and A. Pathania, "Arm-co-up: Arm co operative u tilization of p rocessors," *ACM Transactions on Design Automation of Electronic Systems*, 2024.

[31] H.-Y. Chang, S. H. Mozafari, C. Chen, J. J. Clark, B. H. Meyer, and W. J. Gross, "Pipebert: High-throughput bert inference for arm big. little multi-core processors," *Journal of Signal Processing Systems*, vol. 95, no. 7, pp. 877–894, 2023.

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

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[34] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[35] Y. Wu, M. Schuster, Z. Chen, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[36] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pmlr, 2015, pp. 448–456.

A

Appendix

Table A.1: Multi-head experiment result.

	h	1		2		4		6		8		12	
d_{model}	$\mathcal{L}$	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU
192	16	11.237	1.341	10.968	6.209	8.884	6.019	10.124	4.987	8.814	5.592	9.115	4.935
384	16	15.101	2.215	14.450	8.384	14.620	8.137	14.568	8.491	14.724	8.530	14.073	8.649
576	16	27.343	3.034	26.742	14.731	26.655	14.199	26.857	14.243	26.698	14.266	26.886	14.187
768	16	34.195	3.859	33.195	18.387	33.826	17.576	33.792	18.015	33.323	17.887	33.772	17.647
960	16	42.287	4.684	41.279	22.661	41.261	21.756	41.494	21.876	41.458	21.866	41.209	21.853
192	64	20.852	4.243	20.785	11.837	20.863	11.471	20.859	11.674	20.898	11.593	21.236	11.826
384	64	43.248	8.071	43.410	24.180	43.476	23.559	43.263	23.943	43.718	23.675	43.590	24.081
576	64	73.955	11.761	76.180	40.557	76.086	40.578	76.503	40.907	75.813	40.808	76.486	40.533
768	64	104.441	15.482	108.954	56.868	108.379	57.637	108.481	57.885	108.205	57.482	108.668	57.436
960	64	148.586	19.067	153.421	79.950	153.337	80.503	153.530	81.105	152.987	80.658	153.217	80.426
192	128	41.663	8.999	41.667	23.656	41.562	23.084	41.682	23.704	41.402	23.909	42.639	24.505
384	128	87.996	18.279	87.668	49.836	86.807	48.330	86.262	48.852	86.456	48.309	86.808	49.152
576	128	140.352	26.188	153.195	78.240	153.610	82.468	153.510	83.810	153.816	83.458	155.075	84.246
768	128	212.097	33.992	229.460	116.018	228.898	121.487	228.217	122.754	227.639	121.626	227.828	122.031
960	128	302.603	41.938	323.350	163.597	322.774	169.684	323.457	170.987	323.015	170.007	324.085	170.045
192	256	91.244	22.271	91.185	53.187	91.179	52.344	93.760	54.532	94.389	57.433	95.410	60.411
384	256	185.804	41.248	186.828	105.836	193.623	103.522	195.216	109.649	197.439	110.464	195.973	115.898
576	256	287.121	63.478	338.014	163.307	339.176	184.024	338.316	188.151	340.650	187.243	341.106	192.364
768	256	441.489	86.966	503.974	247.538	503.164	272.318	487.640	276.413	502.466	267.303	505.791	277.196
960	256	626.528	108.073	609.872	344.443	707.423	327.160	695.110	383.538	692.611	378.927	705.706	373.769
192	384	145.000	40.932	146.645	86.721	150.370	86.522	151.192	94.222	147.831	96.239	154.164	95.972
384	384	247.761	75.052	325.684	147.402	326.088	182.956	327.893	188.459	315.437	190.146	334.464	192.417
576	384	447.215	117.015	556.983	258.965	548.489	306.537	558.409	309.100	559.648	313.604	565.049	321.928
768	384	687.575	159.741	675.201	393.518	825.803	375.174	828.354	456.980	788.046	455.924	823.198	441.910
960	384	970.729	196.137	935.191	545.774	1146.926	511.435	1148.400	626.566	1153.617	623.106	1156.217	632.258
192	512	164.679	57.786	160.911	100.038	161.797	98.224	166.214	105.913	174.815	112.200	176.125	126.506
384	512	428.054	123.758	449.282	256.414	433.413	258.775	438.104	261.673	435.415	257.012	445.394	266.126
576	512	626.216	188.130	650.438	373.544	623.840	371.741	623.394	369.803	655.430	370.667	627.082	387.482
768	512	956.041	252.055	947.355	556.941	987.133	534.205	971.699	565.352	942.389	553.250	953.754	551.010
960	512	1348.188	311.734	1282.178	771.726	1368.359	717.064	1340.348	770.133	1374.591	752.115	1333.863	779.321

Table A.2: Feed forward depth experiment result.

	d_{ff}	1920		2304		2688		3072		3584		4096	
d_{model}	$\mathcal{L}$	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU
192	16	0.125	20.202	0.125	12.349	0.126	14.307	0.125	16.791	0.126	16.353	0.126	20.098
384	16	0.125	31.553	0.124	19.330	0.126	22.655	0.124	27.867	0.126	38.365	0.125	40.258
576	16	0.128	43.179	0.126	26.530	0.125	31.982	0.126	39.852	0.126	45.095	0.126	60.000
768	16	0.127	63.298	0.127	39.071	0.128	45.489	0.127	56.551	0.129	66.087	0.131	95.815
960	16	0.128	69.848	0.128	43.987	0.127	50.539	0.128	66.949	0.133	80.038	0.197	113.285
192	64	0.128	21.805	0.128	12.964	0.130	14.897	0.129	17.426	0.129	20.131	0.130	23.642
384	64	0.129	38.441	0.129	23.668	0.131	27.853	0.130	33.631	0.130	38.272	0.132	49.329
576	64	0.131	55.455	0.133	33.485	0.132	38.241	0.132	48.137	0.132	54.608	0.133	74.028
768	64	0.132	75.732	0.134	45.578	0.134	53.409	0.133	67.817	0.147	76.734	0.134	111.308
960	64	0.135	82.993	0.134	52.564	0.135	60.187	0.134	78.295	0.145	92.983	0.191	128.001
192	128	0.132	27.365	0.131	16.886	0.131	17.440	0.131	20.408	0.131	24.159	0.132	28.512
384	128	0.131	44.789	0.133	28.083	0.132	33.389	0.133	39.674	0.133	46.145	0.134	58.151
576	128	0.136	63.699	0.136	39.084	0.136	45.811	0.135	56.510	0.136	65.610	0.139	87.707
768	128	0.137	92.351	0.139	56.594	0.138	64.425	0.139	81.527	0.145	95.724	0.139	128.399
960	128	0.139	101.624	0.139	63.126	0.140	72.802	0.140	93.700	0.152	111.314	0.230	158.692
192	256	0.133	25.845	0.133	16.120	0.133	18.679	0.133	30.772	0.133	37.474	0.133	43.411
384	256	0.137	59.726	0.137	38.732	0.137	47.613	0.137	56.135	0.139	65.458	0.138	75.823
576	256	0.139	88.941	0.140	55.110	0.139	64.699	0.141	80.558	0.141	86.784	0.145	111.715
768	256	0.138	127.889	0.139	77.780	0.141	90.957	0.139	103.682	0.140	121.163	0.141	156.160
960	256	0.141	148.298	0.141	92.049	0.141	96.200	0.139	123.348	0.199	145.744	0.289	199.871
192	384	0.137	34.461	0.137	20.847	0.138	24.322	0.137	29.703	0.138	34.854	0.138	41.176
384	384	0.139	61.871	0.139	38.201	0.139	44.850	0.140	68.109	0.141	79.112	0.141	104.955
576	384	0.141	100.684	0.142	73.343	0.142	84.812	0.141	100.926	0.142	113.726	0.143	152.713
768	384	0.140	160.982	0.141	96.621	0.141	112.044	0.143	131.347	0.143	155.781	0.143	200.979
960	384	0.142	188.231	0.143	116.974	0.142	129.167	0.146	158.669	0.296	191.517	0.931	237.935
192	512	0.137	43.440	0.138	26.400	0.138	30.429	0.139	36.847	0.139	41.864	0.138	50.488
384	512	0.140	78.333	0.141	48.107	0.140	55.558	0.141	67.857	0.140	78.058	0.141	102.235
576	512	0.140	113.186	0.141	67.417	0.140	78.669	0.141	98.097	0.140	110.239	0.141	176.053
768	512	0.140	171.970	0.140	102.919	0.141	119.795	0.141	165.185	0.141	193.319	0.141	250.157
960	512	0.142	198.874	0.142	123.015	0.142	161.128	0.171	198.934	0.304	236.422	6.141	284.048