ECE 479: IoT and Cognitive Computing

Spring 2024, Homework 3

Yanghonghui Chen/ yc47

Please submit your answers by using LaTeX or Word compiled pdf.

Due: April 15th, 11:59 pm

Question 1: GPU and GPU programming (25 pts)

1. List at least three significant differences between GPU and CPU. [**3 pts**]

1. **Architecture and Functionality**: - CPUs are designed for general-purpose computing tasks, such as running applications, managing operating systems, and executing sequential instructions. They typically have a few powerful cores optimized for tasks requiring low-latency processing and complex decision-making. - GPUs, on the other hand, are specialized processors primarily intended for graphics rendering and parallel processing tasks. They contain a large number of smaller, less powerful cores optimized for parallel computations, making them highly efficient for tasks that can be divided into many smaller tasks, such as rendering images, video processing, and machine learning computations.
2. **Parallelism and Execution Model**: - CPUs typically have a few cores (ranging from 2 to 64 in consumer-grade CPUs) optimized for executing sequential instructions in a linear fashion. They excel at tasks that require high single-threaded performance and complex decision-making, such as running operating systems and applications. - GPUs are massively parallel processors with hundreds to thousands of cores. They excel at tasks that can be parallelized across many cores, such as rendering graphics, processing large datasets, and running deep learning algorithms. GPUs use a SIMD (Single Instruction, Multiple Data) execution model, where each core performs the same operation on multiple data elements simultaneously, allowing for highly parallel computation.
3. **Memory Hierarchy and Bandwidth**: - CPUs typically have a small amount of high-speed cache memory (L1, L2, L3 caches) integrated into the processor die, along with access to system memory (RAM) via a memory controller. They

prioritize low-latency access to data, making them suitable for tasks that require frequent data access and manipulation. - GPUs have larger onboard memory (VRAM) compared to CPUs, optimized for high-throughput data transfer and processing. This memory is shared among all cores and is accessed through high-speed memory buses. While GPUs sacrifice some latency for increased bandwidth, they excel at streaming large datasets and performing memory-bound computations, such as texture mapping in graphics rendering and matrix operations in deep learning.

2. Consider the image blurring algorithm in lecture 15, slide 44. Answer the following questions.

   (a) To blur a monochrome image with the size of $(600, 400)$, calculate the Grid dimension for the kernel given the block size of $(16, 16, 1)$. **[3 pts]**

   For the image size of (600, 400), and assuming each block covers a 16x16 pixel region:

   - Number of blocks in the x-direction = ceil(600 / 16) = 38
   - Number of blocks in the y-direction = ceil(400 / 16) = 25

   So, the Grid dimension for the kernel would be (38, 25, 1).

   (b) Does the `BLUR_SIZE` have effects on your calculated Grid dimension? Justify your answer.**[3 pts]**

   The BLUR SIZE affects the calculated Grid dimension indirectly because it determines the number of iterations required in the algorithm. However, the block size remains constant at (16, 16, 1). Therefore, the Grid dimension would remain the same regardless of the BLUR SIZE.

   (c) If `BLUR_SIZE` is 2, and the height and width of the image are much larger than `BLUR_SIZE`. How many different possible values could variable `pixels` take right before line 10 (exit the for loop) of the algorithm? Hint: consider different boundary conditions **[3 pts]**

   1. Periodic Boundary Conditions: With periodic boundary conditions, the image wraps around at the edges. Therefore, there could be an infinite number of valid pixels within the `BLUR_SIZE x BLUR_SIZE` box, leading to

an infinite number of possible values for `pixels`.
2. Reflective Boundary Conditions: In this case, the number of valid pixels within the `BLUR_SIZE x BLUR_SIZE` box would depend on the distance from the edge of the image. Pixels near the edges would have fewer valid neighbors, limiting the possible values for `pixels` but not necessarily leading to a finite number of values.
3. Zero-padding or other Boundary Conditions: With zero-padding or similar conditions, the number of valid pixels within the `BLUR_SIZE x BLUR_SIZE` box would be limited by the size of the image and the `BLUR_SIZE`. This would result in a finite number of possible values for `pixels`.

3. Parts of a GPU: write down the letter that describes the part **[8 pts]**

   A. A group of threads (typically 32 for NVIDIA) executed concurrently

   B. The smallest unit of execution

   C. The largest memory pool available in a GPU, accessible by all threads

   D. Specialized hardware components designed to accelerate mixed-precision matrix arithmetic

   E. General-purpose, high-speed serial bus standard widely used in computers to connect various peripherals

   F. A smaller, faster memory pool located within each SM, accessible by all threads within a single thread block executing on that SM

   G. The highest level of abstraction for organizing threads in a GPU kernel launch

   H. A larger group of threads that enables flexible synchronization and scheduling

   I. Point-to-point interconnect technology that enables faster communication between multiple GPUs or between GPUs and CPUs

   J. The fundamental building block of a GPU, consisting of multiple processing cores

   (G) Grid

   (H) Block

   (A) Warp

   (B) Thread

   (D) Tensor Core

   (I) NVLink

   (F) Shared Memory

   (C) Global Memory

4. Please review the concepts regarding GPU programming and embedded GPU. For each of the following statements, write down True or False according to your best knowledge. To receive full credit for each False statement you identify, you need to justify your answer. **[5 pts]**

   (a) GPU is winning CPU in all scenarios thanks to its massive number of execution units with a zero-latency context switch.

   (a) False. While GPUs excel in parallel processing tasks, they may not outperform CPUs in all scenarios. CPUs are still more efficient for sequential tasks and those requiring low latency context switches due to their design optimized for single-threaded performance

   (b) Threads are grouped into blocks for easy managing and better cooperation, and thus blocks are scheduling and execution units in GPU architecture.

   (b) False. Threads are grouped into warps for scheduling and execution, and warps are the scheduling and execution units in GPU architecture. Blocks are collections of threads managed and scheduled together.

   (c) By mapping an algorithm onto the GPU, we could decrease the computational complexity big-$\mathcal{O}$ of the algorithm by exploiting the parallelism in the algorithm and the hardware.

   (c) True. By leveraging the parallelism offered by GPUs, algorithms can often achieve a decrease in computational complexity, as the workload can be distributed across multiple processing units, reducing the overall runtime.

   (d) Tensor Cores accelerate training and inference of DNN by massive concurrent Multiply-Add operations.

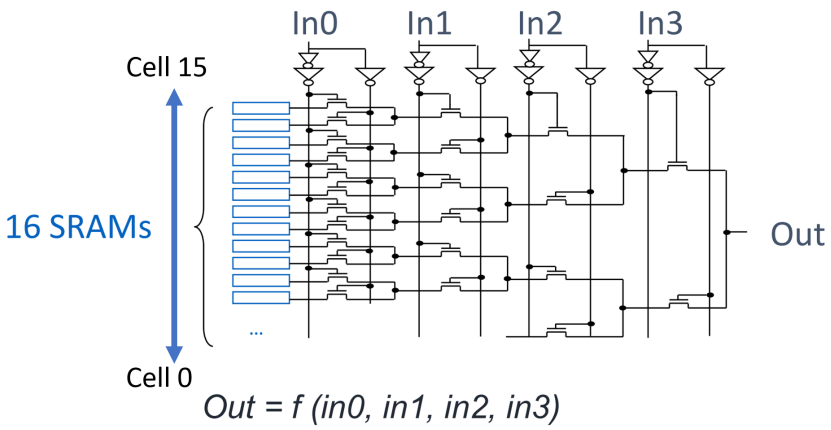
(d) True. Tensor Cores are specialized hardware units designed to accelerate matrix multiplication operations commonly used in deep neural network (DNN) training and inference. They achieve this acceleration by performing massive concurrent Multiply-Add operations.

(e) Similar to desktop-level (full-powered) GPUs, mobile (embedded) GPUs like Jetson Xavier NX also suffer latency concerns while copying data between host and device.

(e) True. Similar to desktop-level GPUs, mobile/embedded GPUs like the Jetson Xavier NX also face latency concerns when copying data between the host and device, although they may employ specialized optimizations to mitigate these concerns to some extent.

Question 2: FPGA Basics and High-Level Synthesis (25 pts)

1. Suppose that we have a simple 4-bit look-up table as shown below (only the upper 12 SRAM cells are shown):



(a) Briefly explain why this LUT can work as any 4-bit binary logic function. (4-bit input and 1-bit output) **[2 pts]**

Answer: This 4-bit look-up table (LUT) essentially functions as a memory unit that stores predefined outputs for all possible combinations of 4-bit inputs. The key to its versatility lies in the concept of programmability. Each of the 16 possible input combinations (from 0000 to 1111 in binary) corresponds to a unique output value, which can be programmed into the memory cells.

When you input a specific 4-bit binary value, the LUT retrieves the corresponding output value from its memory cells and provides it as the output. In other words, the LUT essentially "looks up" the output value associated with the input pattern stored in its memory.

This functionality allows the LUT to emulate any 4-bit binary logic function because you can program it with the desired output values for each input combination. Whether it's a simple AND, OR, XOR, or any complex logic function, as long as you provide the correct output values for each input pattern, the LUT will effectively perform the desired logic function.

In essence, the LUT acts as a flexible, programmable logic element, capable of implementing various logic functions by storing and retrieving the appropriate output values based on the input pattern provided. This versatility makes it a fundamental building block in digital circuit design and implementation.

(b) Pass transistors in FPGAs are used as electronic switches that control the connection paths between different logic blocks (like LUTs) and I/O blocks within the FPGA. They are called "pass" because they allow signals to "pass" through them when turned on (conducting) and block signals when turned off (non-conducting). Suppose that we want to access cell 9 of this LUT. Please write down the input to the four input bits (MSB-LSB [in3 in2 in1 in0]). Also highlight the activated pass transistors on the path to the output in the diagram. **[2 pts]**

MSB-LSB [in3 in2 in1 in0] = [1 0 0 1]

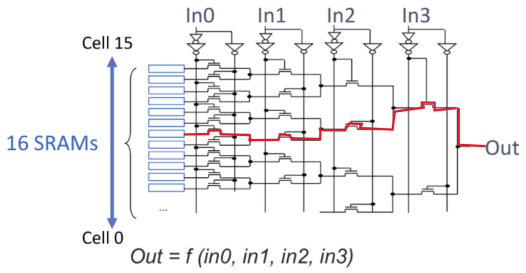


Figure 1: Activated Pass Transistors on the Path

2. In the following toy HLS C code snippet, there are multiple opportunities to apply the optimization techniques. Apply at least 2 of them and write down the line number where you can insert the pragma and explain the effect of inserting these pragmas. Describe in one or two sentences if you need to change the code structure. [**4 pts**]

```
1   void top(){
2     int array[100];
3     foo(array);
4     bar(array, 20);
5     return;
6   }
7
8   void foo(int array[100]){
9     for (int i=0;i < 100;i ++){
10      array[i] = 0;
11    }
12    return;
13  }
14
15  void bar(int array[100], int boundary){
16    for (int i=0;i < 100;i ++){
17      array[i] = i;
18    }
19    for (int i=50;i < 100;i ++){
20      array[i] *= 2;
21    }
22
23    for (int i=0;i < boundary;i ++){
24      array[i] += 10;
25    }
26    return;
27  }
28
```

(a) Loop Fusion:

- Line Number: 16 to 24
- Pragma: `#pragma HLS LOOP_MERGE force`
- Effect:

  Loop fusion combines multiple loops into a single loop, reducing loop overhead and improving cache utilization.In this case, fusing the loop that adds 10 to array elements (`for (int i = 0; i < boundary; i++)`) with the loop that initializes the array elements (`for (int i = 0; i < 100; i++)`) reduces the overall number of loop iterations and loop overhead. This can lead to better performance and resource utilization, especially in FPGA implementations where resources are limited. Modification of Codes from line 16-24:

```
    for (int i = 0; i < 100; i++) {
        array[i] = i;
        if (i >= 50) {
            array[i] *= 2; // Doubling values for index >= 50
        }
        if (i < boundary) {
            array[i] += 10; // Adding 10 to the first 'boundary' elements
        }
    }
    return;
}
```

(b) Loop Pipelining:

- Line Number: 23
- Pragma: `#pragma HLS PIPELINE II`
- Effect: Loop pipelining can be applied to the loops to optimize parallelism. In this case, we can apply it to the loop in the `bar` function that initializes the array elements. Since this loop is the most computationally intensive part, pipelining it can potentially improve performance.

3. In the following HLS C code for a convolution, there is a pipelining pragma at line 10. Read the code and answer the following questions.

```
1  void conv3x3(
2      int32 feature[3][112][112],
3      int32 kernel[32][3][3][3],
4      int32 output[32][110][110])
5  {
6      for(int x_out = 0; x_out < 110; x_out ++) {
7          for(int c_out = 0; c_out < 32; c_out ++) {
8              for(int y_out = 0; y_out < 110; y_out ++) {
9  #pragma HLS PIPELINE
10                 int32 acc = 0;
11                 for(int c_in = 0; c_in < 3; c_in ++) {
12                     for (int y_k = 0; y_k < 3; y_k ++) {
13                         for (int x_k = 0; x_k < 3; x_k ++) {
14                             acc += kernel[c_out][c_in][y_k][x_k]
       * feature[c_in][y_k + y_out][x_k + x_out];
15                         }
16                     }
17                 }
18                 output[c_out][y_out][x_out] = acc;
19             }
20         }
21     }
22 }
23
```

(a) Explain what is "II" when we talk about pipelining. Can we achieve II = 1 in this code if no other optimization is done? Why or why not? Assuming the memory has two ports for read and write. (Hint: How many concurrent memory access do we need to achieve II = 1?) **[3 pts]**

(a) In the context of pipelining, II (Initiation Interval) represents the number of clock cycles between the start of consecutive loop iterations. To achieve II = 1 in the provided loop, we need to ensure that all memory accesses required by each iteration of the loop can occur concurrently. Let's analyze the memory accesses in the loop:

i. `feature` array: Each iteration accesses `feature[c_in][y_k + y_out][x_k + x_out]`.

ii. `kernel` array: Each iteration accesses `kernel[c_out][c_in][y_k][x_k]`.

For II = 1, we need to ensure that all memory accesses are independent and can occur simultaneously. In this case, we need:

- Concurrent read access to the `feature` array for `feature[c_in][y_k + y_out][x_k + x_ou`
- Concurrent read access to the `kernel` array for `kernel[c_out][c_in][y_k][x_k]`.

Since both `feature` and `kernel` arrays are read-only in this loop, we need at least two read ports for each array to achieve II = 1. This would allow us to simultaneously read from different locations in both arrays in each clock cycle, satisfying the memory access requirements for the loop iterations. However, currently the memory has only two ports for read and write. So it is not possible in this code without further optimization.

(b) Given the position of this pipelining pragma, what happens to the inner three nested loops below the `#pragma HLS PIPELINE`? [**2 pts**]

(b) With the pipelining pragma applied at line 9, the inner three nested loops are pipelined. This means that the iterations of these loops can execute concurrently, with each iteration starting after the previous one has advanced to the next stage of computation.

(c) After the application of the `PIPELINE` pragma, draw the datapath that would compute the value `acc` (hint: added tree). Assume the adder tree can be pipelined, how many pipeline stages will the adder tree have? Assume one multiplication and one addition will each take one cycle to compute. How many cycles are needed to complete the `acc` computation? [**4 pts**]

(c) Since we have three nested loops with each loop having three iterations (`for (int c_in = 0; c_in < 3; c_in++)`, `for (int y_k = 0; y_k < 3; y_k++)`, and `for (int x_k = 0; x_k < 3; x_k++)`), the adder tree will have a depth of 3 (corresponding to the innermost loop).

Therefore, the number of cycles needed to complete the `acc` computation will be equal to the depth of the adder tree, which is 3 cycles.

(d) If we achieve II = 2 in this function, assume the assignment on line 9 takes one cycle, the adder tree for the `acc` computation takes the number of cycles derived from (c) above, the assignment on line 18 takes one cycle, and there are no other latency overheads. What is the total latency of this convolution function? **[6 pts]**

(d) If we achieve II = 2 in this function, it means that each iteration of the outer loop (line 6) takes 2 cycles to execute. Let's break down the latency for each part of the computation:

i. The assignment on line 9 (`int32 acc = 0;`) takes one cycle.

ii. The adder tree for the `acc` computation takes 3 cycles, as determined in the previous answer.

iii. The assignment on line 18 (`output[c_out][y_out][x_out] = acc;`) takes one cycle.

Considering that the loop (line 6) takes 2 cycles per iteration, the total latency for each iteration of the loop will be the sum of the latencies of these operations:

Total latency per iteration = Assignment on line 9 + Adder tree cycles + Assignment on line 18 = 1 cycle + 3 cycles + 1 cycle = 5 cycles. Here,

- D is the pipeline depth (latency) per iteration,
- II is the initiation interval of the loop,
- N is the number of iterations.

We substitute the given values: D = 5 cycles (latency per iteration), II = 2 (initiation interval of the loop), N = 110 (number of iterations).

Total latency = II * N + D - II = 2 * 110 + 5 - 2 = 220 + 5 - 2 = 223 cycle

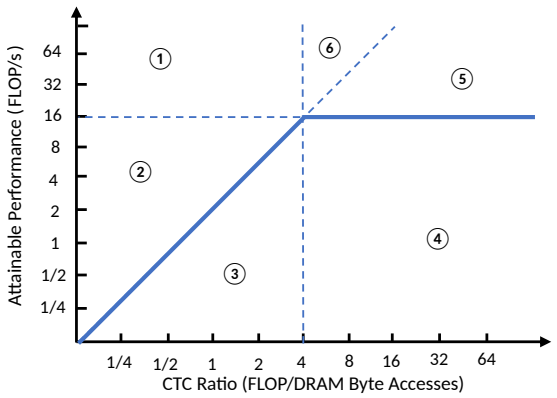
(e) If we want to achieve II = 1, what other optimization technique do we

must perform? **[2 pts]**

(e) Introducing array partitioning involves dividing the `kernel` and `feature` arrays into smaller partitions, enabling concurrent accesses and reducing contention for memory ports. By partitioning along appropriate dimensions, such as channel or spatial dimensions, we can increase memory access parallelism and reduce dependencies between iterations. The optimal partition size needs to be determined based on memory access patterns and hardware constraints, ensuring efficient utilization of resources and achieving performance improvements. Overall, array partitioning effectively addresses the port limit issue and enhances the efficiency and performance of the computation in the provided code.

Question 3: Efficient DNN Accelerator (15 pts)

1. **Roofline Model**
   Suppose that a naive accelerator design falls in region 2 of the roofline in the following diagram.



(a) Is this a feasible design? Why or why not? Which regions indicate a feasible design? **[2 pts]**

(a) A design falling in region 2 of the roofline model, on the left side of the curve, may not be feasible because it indicates that the performance achieved is limited by memory bandwidth rather than compute capability. In this region, the compute capability of the accelerator exceeds the memory bandwidth available, leading to inefficient utilization of compute resources due to memory access bottlenecks. Feasible designs typically fall in regions where the performance is limited by compute capability, such as regions above and to the right of the curve.

(b) What does it mean when the graph "flats out" after the CTC ratio is greater than 4 FLOP/DRAM? **[2 pts]**

(b) When the graph "flats out" after the CTC ratio is greater than 4 FLOP/-DRAM, it indicates that increasing computational intensity (FLOP/DRAM) beyond this point does not result in a significant improvement in performance. This suggests that the memory system has become saturated, and further increasing computational intensity does not lead to a proportional increase in performance. In other words, the accelerator's performance is no longer limited by memory bandwidth but by other factors such as compute capability or architectural limitations.

(c) What is the key challenge that this design is facing in terms of improving performance? Propose a possible technique to solve this challenge. [**2 pts**]

(c) The key challenge that this design is facing in terms of improving performance is the imbalance between compute capability and memory bandwidth. To address this challenge, a possible technique is to optimize memory access patterns and data movement to improve memory bandwidth utilization. This can involve techniques such as data prefetching, data reuse, memory locality optimizations, and memory hierarchy optimizations. By reducing memory access latencies and maximizing memory bandwidth utilization, the design can achieve better performance and potentially move to a region of the roofline model where compute capability is the limiting factor.

2. **Depthwise-pointwise Convolution**

   (a) Standard convolution takes an $h_i \times w_i \times d_i$ (representing height, width, and number of channels) input tensor L_i, and applies convolutional kernel $K \in R^{k \times k \times d_i \times d_j}$ to produce an $h_i \times w_i \times d_j$ output tensor L_j with $stride = 1$. Please compute the number of operations (regarding both multiplications and additions). **[2 pts]**

   (a) The number of operations (both multiplications and additions) for one output element is given by: Operations per output element=($k \times k \times di \times$channels in kernel)+($k \times k \times di \times$(channels in kernel1))

   This includes the multiplications required for computing the dot product and the additions required for summing up the results.

   Therefore, the total number of operations for the entire convolution operation is: Total operations=Operations per output element×Total output elements.

   Substituting the expressions for operations per output element and total output elements, we get: Total operations=((k×k)+(k×k))×(hi-k+1))×(wi-k+1)×dj×di

   (b) Depthwise convolution takes an $h_i \times w_i \times d_i$ input tensor L_i, and applies convolutional kernel $K \in R^{k \times k \times d_i}$ to produce an $h_i \times w_i \times d_i$ output tensor L_j with $stride = 1$. Please compute the number of operations (regarding both multiplications and additions). **[2 pts]**

   (b) Total operations=((k×k)+(k×k))×(hi-k+1))×(wi-k+1)×di

   (c) Pointwise convolution takes an $h_i \times w_i \times d_i$ input tensor L_i, and applies convolutional kernel $K \in R^{1 \times 1 \times d_i \times d_j}$ to produce an $h_i \times w_i \times d_j$ output tensor L_j with $stride = 1$. Please compute the number of operations (regarding both multiplications and additions). **[2 pts]**

   (c) Total operations=2×hi×wi×di×dj

(d) By combining the depthwise and pointwise convolutions, we can replace the standard one as described in Lecture 19 page 44. Compute the number of operations in a depthwise convolution and a pointwise convolution in (b) and (c), and compare it to the operation number of a standard convolution (a). What is the ratio of operation reduction if using a depthwise separable convolution instead of a standard one? Please explain why it can save operations. **[3 pts]**

(d) Operations saved=Total operationsstandard-(Total operationsdepthwise+Total operationspointwise)

Then, we can calculate the ratio of operation reduction as follows:

Ratio of operation reduction=Operations saved / Total operationsstandard×100%.

Depthwise separable convolution saves operations by reducing the number of parameters and computations compared to standard convolution. It achieves this by first performing a depthwise convolution, which computes spatial features independently across each input channel, and then a pointwise convolution, which mixes the spatial features through a 1x1 convolution across channels. By decomposing the standard convolution into depthwise and pointwise convolutions, depthwise separable convolution reduces the number of parameters and computations, leading to fewer operations and increased computational efficiency. This approach is particularly effective when the input tensor has a large number of channels, as it significantly reduces the computational cost while maintaining or even improving performance.

Question 4: Attention is all you need (15 pts)

1. Describe one similarity and one difference between Computer Vision tasks and Natural Language Processing tasks in machine learning. **[4 pts]**

   (a) One similarity between Computer Vision (CV) tasks and Natural Language Processing (NLP) tasks in machine learning is that both deal with complex input data that contain rich semantic information. In CV tasks, input data typically consists of images or videos, where each pixel or frame encodes visual information such as shapes, textures, and colors. Similarly, in NLP tasks, input data consists of text sequences, where each word or token carries semantic meaning. Both CV and NLP models aim to understand and extract meaningful information from these data sources to solve specific tasks such as image classification, object detection, sentiment analysis, or machine translation.

   (b) One difference between CV and NLP tasks lies in the nature of the input data and the methods used to process them. In CV tasks, the input data are typically structured in a grid-like format, such as images represented as 2D arrays of pixels. Convolutional neural networks (CNNs) are commonly used in CV tasks to extract hierarchical features from local image patches and capture spatial relationships effectively. On the other hand, in NLP tasks, the input data are sequential and unordered, such as sentences or paragraphs represented as sequences of tokens. Recurrent neural networks (RNNs) and transformer-based architectures, such as the Transformer and BERT, are commonly used in NLP tasks to capture sequential dependencies and contextual relationships between words or tokens in the input text. Therefore, while both CV and NLP tasks aim to understand complex data sources, they require different modeling approaches and architectures due to the distinct nature of their input data.

2. When constructing the Q-K matrix, we find that the Query has a dimension of $(64, 1024, 25)$ (batch size, channel depth, sequence length).

   (a) We have 32 heads in the multi-head attention structure. What are the dimensions of Q fed into each attention module after the head-split? **[1 pts]**

   (64,32,32,25)

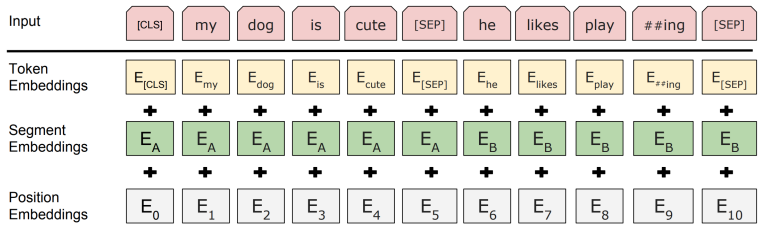
(b) What is the dimension of the Q-K matrix in each attention module? **[1 pts]**

(64,32,25,25)

(c) Think about the matrix multiplication operation and the number of the Q-K matrices produced by the head-split. Explain why multi-head is necessary. **[3 pts]**

In multi-head attention, each head learns different relationships between words in the input sequence. By having multiple heads, the model can attend to different parts of the input sequence simultaneously, capturing various aspects of the data. This not only increases the model's capacity to capture complex patterns but also helps to mitigate the limitation of single attention mechanism by allowing the model to focus on different aspects of the input, leading to improved performance.

3. Referring to the following picture about BERT, answer the following questions.



(a) Explain in two sentences how BERT training differs from GPT at the time. **[2 pts]**

Regarding BERT training versus GPT at the time, BERT employs a bidirectional training objective, where the model is trained to predict masked words in a sentence, enabling it to capture contextual information from both directions of the text. This differs from GPT, which utilizes a unidirectional autoregressive language modeling objective, where the model predicts the next word in a sequence based on the preceding context. Thus, BERT can effectively capture bidirectional context, while GPT primarily focuses on generating text in a forward direction.

(b) How does the "Segment Embeddings" help BERT train? **[2 pts]**
Segment embeddings in BERT assist in training by providing the model with contextual information about the different segments or sentences within the input sequence. By assigning unique segment IDs to tokens from different segments and incorporating corresponding segment embedding vectors, BERT can differentiate between segments during training. This enables the model to effectively capture relationships and dependencies between tokens from different segments, facilitating tasks that involve understanding interactions across multiple segments, such as sentence-pair classification or question answering. Overall, segment embeddings enhance BERT's ability to learn rich representations of text by incorporating information about the structure and context of input sequences.

(c) Why do Transformer models need "Position Embeddings"? **[2 pts]**

Transformer models need position embeddings to encode the positional information of tokens within the input sequences. Unlike recurrent neural networks (RNNs) or convolutional neural networks (CNNs), transformers process tokens in parallel without considering their sequential order. Position embeddings provide a mechanism for the model to understand the relative positions of tokens in the sequence, allowing it to capture the sequential relationships and dependencies crucial for tasks like natural language understanding and generation. This ensures that the model can effectively process and generate coherent outputs based on the order of tokens in the input sequence.

4. **(Bonus)** Find an online AI Image generator, such as Microsoft Designer, create a picture with your favorite animal doing some human activities. Post your picture here and on Campuswire, including your prompt and the artwork. The bonus score is given based on your creativity. **[5 pts]**

Figure 2: Prompt: An astronaut panda in a rocket to be launched is waving at the crowd on the ground

Question 5: IoT Security and Cybersecurity (20 pts)

1. In the lecture, we have introduced three security solutions proposed for Intel, ARM, and RISC-V architectures. The common strategy, on which these approaches all focus, is *Isolation*. Pick one scenario from Intel SGX, ARM TrustZone, and RISC-V Keystone, and on a high level, explain in roughly three sentences how they implement this strategy. [**9 pts**]

   In Intel SGX (Software Guard Extensions), the scenario involves protecting sensitive code and data within a secure enclave, ensuring confidentiality and integrity even in the presence of malicious software or privileged attackers. SGX achieves this by creating isolated memory regions called enclaves, which are encrypted and authenticated by hardware. Only authorized code running within the enclave can access the protected data, while external software or even the operating system cannot tamper with or observe the enclave's contents.

   In ARM TrustZone, the scenario involves securing sensitive operations on a mobile device, such as authentication or payment processing, from potential attacks or malware. TrustZone creates a secure world and a normal world, running concurrently on the same processor but isolated from each other. Sensitive operations are executed in the secure world, protected by hardware-enforced isolation, while the normal world handles non-sensitive tasks, ensuring that critical operations are shielded from unauthorized access or tampering.

   In RISC-V Keystone, the scenario involves safeguarding critical systems in embedded and IoT devices from cyber threats, ensuring the integrity and confidentiality of sensitive data and operations. Keystone implements a secure enclave architecture similar to Intel SGX, leveraging hardware-based memory protection mechanisms to isolate trusted execution environments (TEEs). These TEEs provide a secure execution environment for critical applications, protecting

them from unauthorized access or tampering by malicious software or external adversaries.

2. Many edge devices operate on battery power. They are usually in sleep mode when no data processing tasks are needed. Under such an energy constraint, the device is prone to Denial-of-Sleep attacks from malicious nodes. Consider a simple IoT device that operates on battery power. To avoid the Man-in-the-Middle attack where a malicious node intercepts the packet halfway in the transmission, the designers of this IoT device have decided to encrypt both the data and the header. Upon receiving the packets, the device first decrypts the header to identify whether the packet comes from a trusted node. If not, the packet is discarded.

   (a) Explain in one sentence how Denial-of-Sleep attack affects an IoT edge device. What specifically does it attack? **[3 pts]**

   A Denial-of-Sleep attack on an IoT edge device disrupts its normal operation by continuously waking it from sleep mode, draining its battery rapidly and preventing it from conserving energy, which compromises its ability to perform essential functions and reduces its operational lifespan.

   (b) As the attacker, you know how to send an encrypted packet to the device, but you cannot authenticate yourself as a trusted node after the device has decrypted your packet. Nevertheless, you can still launch a Denial-of-Sleep attack from your malicious node. Briefly explain how you can conduct this attack. Hint: encryption and decryption data can consume a considerable amount of energy. **[2 pts]**

   As the attacker, even though you cannot authenticate yourself as a trusted node after the device decrypts your packet, you can still launch a Denial-of-Sleep

attack by sending a flood of encrypted packets to the device. Each time the device receives and decrypts a packet, it consumes a significant amount of energy due to the encryption and decryption process. By continuously bombarding the device with encrypted packets, you can force it to repeatedly wake up from sleep mode, leading to rapid battery depletion and ultimately causing the Denial-of-Sleep attack.

3. In the lectures, we have discussed one specific Cyber-attack malware called Mirai. It is used in some massive DDoS attacks.

   (a) What is a DDoS attack? How does it differ from regular DoS attacks? [**2 pts**]

   A DDoS (Distributed Denial of Service) attack is a malicious attempt to disrupt the normal traffic of a targeted server, service, or network by overwhelming it with a flood of internet traffic from multiple sources. Unlike regular DoS attacks, where a single attacker disrupts the target by flooding it with traffic from one source, DDoS attacks involve multiple attackers, often from various locations or compromised devices, coordinating to flood the target with an excessive amount of traffic. This distributed nature makes DDoS attacks more challenging to mitigate and can result in more severe disruptions.

   (b) On a high level, what devices does Mirai take advantage of? [**2 pts**]

   On a high level, Mirai takes advantage of Internet of Things (IoT) devices, such as routers, IP cameras, and DVRs (Digital Video Recorders), which are often poorly secured and easily compromised due to default or weak passwords. Mirai scans the internet for vulnerable IoT devices and infects them, turning them into bots that can be controlled remotely by the attacker to participate in DDoS attacks. These compromised IoT devices form a botnet, which amplifies the attack's scale and impact.

   (c) We know that Mirai is badly written. How does it succeed in infecting many devices? [**2 pts**]

   Mirai succeeds in infecting many devices primarily due to two main factors: its ability to exploit common vulnerabilities found in poorly secured IoT devices and its self-propagation mechanisms. Firstly, Mirai scans the internet for devices with default or weak credentials, exploiting known vulnerabilities to gain unauthorized access. Secondly, once a device is compromised, Mirai spreads rapidly by scanning for other vulnerable devices within the same network or across the internet, using techniques such as brute-force attacks or exploiting known exploits. Additionally, Mirai's widespread propagation

is facilitated by the large number of vulnerable devices connected to the internet and the lack of robust security measures implemented by device manufacturers or users.