ECE 277 - GPU Programming
Final Project Fall 2023

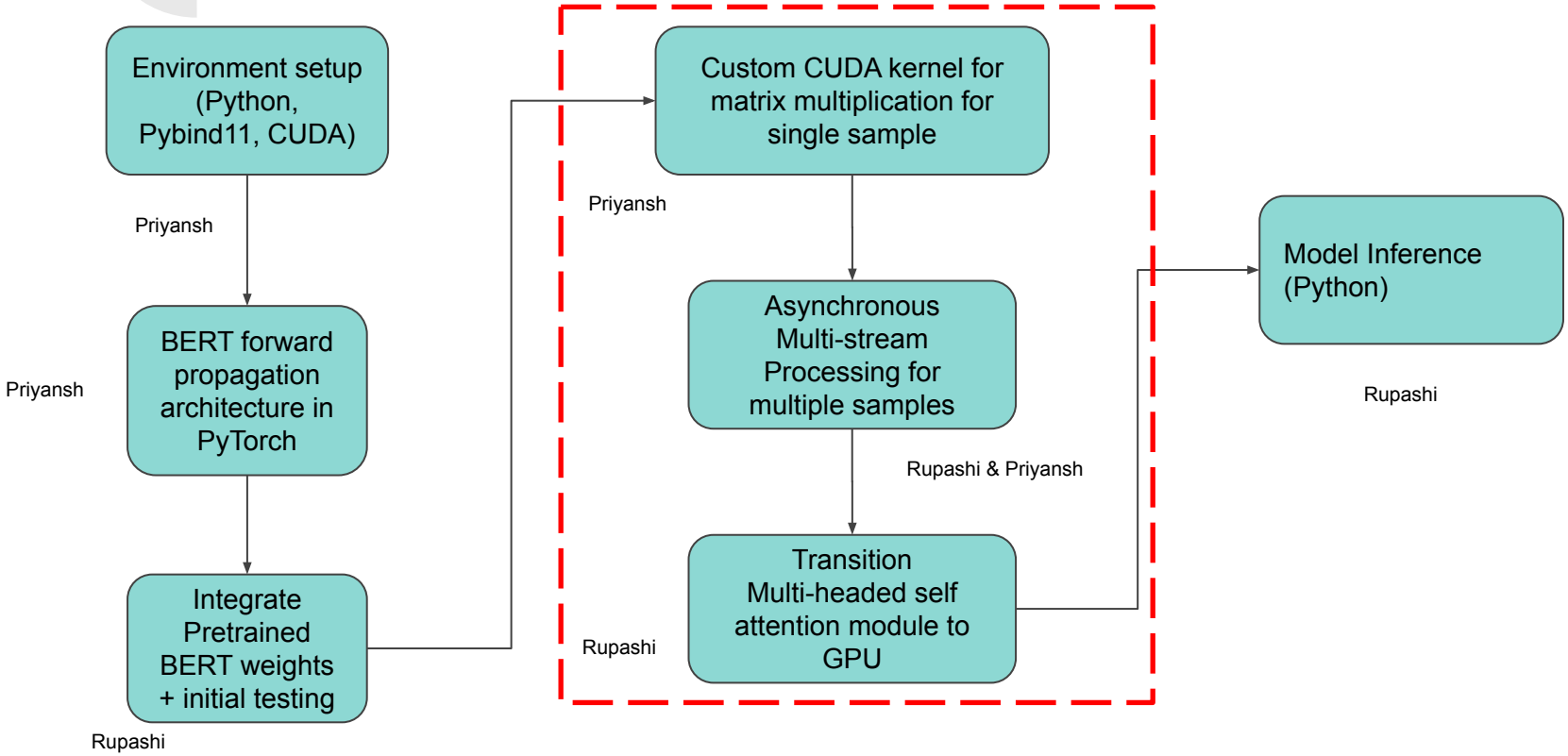
Accelerating Inference Time in NLP Transformer BERT model using PyTorch & CUDA

Submitted by-
Rupashi Sangal : A59019496
Priyansh Bhatnagar: A59019463

Objective

*Develop a custom **CUDA** kernel for efficient 3D matrix multiplication, and integrate it into Python using **Pybind11**, aiming to enhance **BERT** model inference performance.*

Project Framework/ Individual Contributions



Environment Setup

Environment Setup

Data

URL:
https://github.com/MegEngine/Models/blob/master/official/nlp/bert/glue_data/MRPC

The GLUE MRPC (Microsoft Research Paraphrase Corpus) dataset is a part of the GLUE benchmark, consisting of sentence pairs labeled for semantic equivalence, aimed at evaluating models on the task of identifying paraphrases.

Quality	#1 ID	#2 ID	#1 String	#2 String
1	702876	702977	Amrozi accused	Referring to him as only " the
0	2108705	2108831	Yucaipa owned I	Yucaipa bought Dominick 's in
1	1330381	1330521	They had publish	On June 10 , the ship 's owner
0	3344667	3344648	Around 0335 GM	Tab shares jumped 20 cents ,
1	1236820	1236712	The stock rose $	PG & E Corp. shares jumped $
1	738533	737951	Revenue in the fi	With the scandal hanging over
0	264589	264502	The Nasdaq had	The tech-laced Nasdaq Comp
1	579975	579810	The DVD-CCA th	The DVD CCA appealed that c
0	3114205	3114194	That compared v	Earnings were affected by a n

Environment

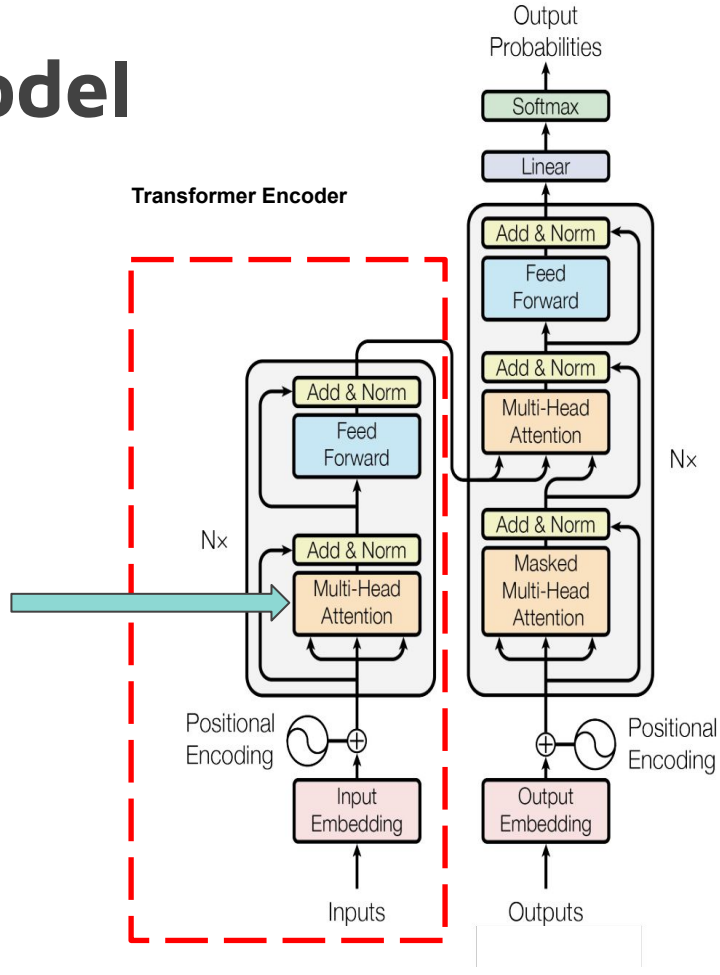
Model Architecture ——————> Python, PyTorch

Model inference ——————> Custom CUDA / C++, PyBind11

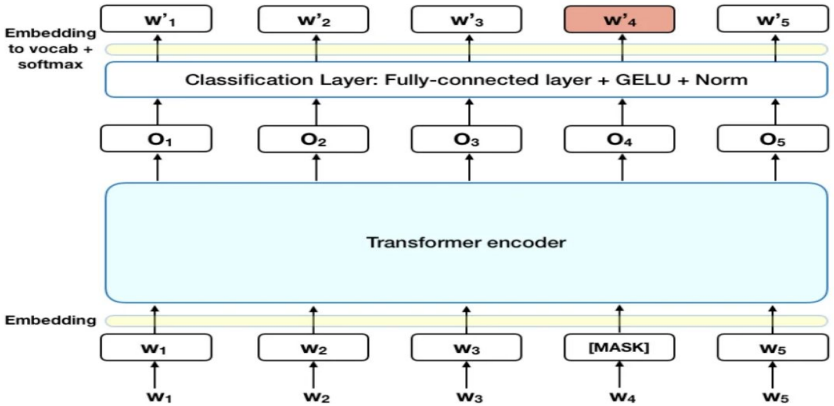
BERT Model

BERT Language Model

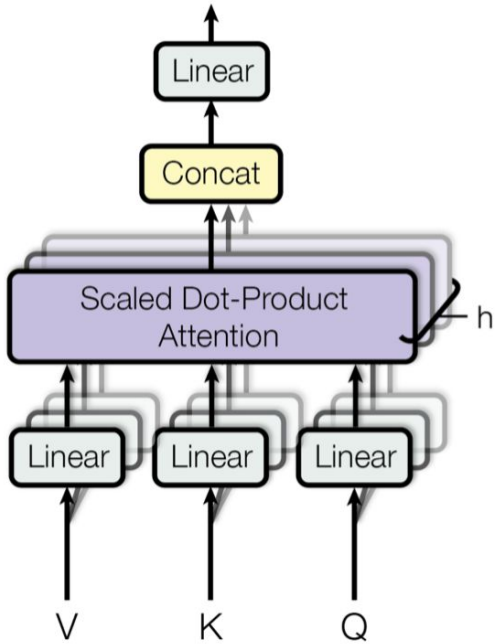
BERT uses a transformer encoder architecture, processing text bidirectionally to capture rich contextual relationships between words.



Model Structure Used



- Input Size (Max sequence) length): 128 tokens
- Embedding Size: 768
- **Number of layers: 12**
- Hidden layer size: 768
- Number of Attention Heads:12
- Size of the feed-forward layers in each transformer block: 3072
- **Total Parameters: 110M**



Multi-Head Attention Module

CUDA Implementations

Data Flow

```python
class MultiHeadedSelfAttention(nn.Module):
    """ Multi-Headed Dot Product Attention """
    def __init__(self, cfg):
        super().__init__()
        self.proj_q = nn.Linear(cfg.dim, cfg.dim)
        self.proj_k = nn.Linear(cfg.dim, cfg.dim)
        self.proj_v = nn.Linear(cfg.dim, cfg.dim)
        self.drop = nn.Dropout(cfg.p_drop_attn)
        self.scores = None # for visualization
        self.n_heads = cfg.n_heads

    def forward(self, x, mask):

        # GPU Accelerated Implementation
        data = x.view(-1, 128*768)
        data_np = data.numpy()

        q_weight = self.proj_q.weight.detach().numpy()
        k_weight = self.proj_k.weight.detach().numpy()
        v_weight = self.proj_v.weight.detach().numpy()

        q_weight_transpose = q_weight.T
        k_weight_transpose = k_weight.T
        v_weight_transpose = v_weight.T

        q = cu_multiply_matrix.multiply(data_np, q_weight_transpose)
        k = cu_multiply_matrix.multiply(data_np, k_weight_transpose)
        v = cu_multiply_matrix.multiply(data_np, v_weight_transpose)

        q_reshaped = q.reshape(-1,128, 768)
        k_reshaped = k.reshape(-1,128, 768)
        v_reshaped = v.reshape(-1,128, 768)

        q = torch.tensor(q_reshaped)
        k = torch.tensor(k_reshaped)
        v = torch.tensor(v_reshaped)
```

Transformer

```cpp
PYBIND11_MODULE(cu_multiply_matrix, m) {
    m.def("multiply", &mm_wrapper, "Linear layer");

#ifdef VERSION_INFO
    m.attr("__version__") = VERSION_INFO;
#else
    m.attr("__version__") = "dev";
#endif
}
```

Pybind11 call

```cpp
__global__ void kernel_multiply(float* A, float* B, float* C, int M, int N, int K)
{
    int I = blockIdx.y * blockDim.y + threadIdx.y;
    int J = blockIdx.x * blockDim.x + threadIdx.x;

    if ((I < N) && (J < M))
    {
        {
            float _c = 0;
            for (unsigned int k = 0; k < K; k++)
            {
                float a = A[I * K + k];
                float b = B[k * M + J];
                _c += a * b;
            }
            C[I * M + J] = _c;
        }
    }
}
```

Kernel function

```cpp
void cu_multiply(float* A, float* B, float* C, int M, int N, int K, int X)
{
    // Assume A = XxNxK, B = KxM matrix, C = XxNxM
    float* d_a, * d_b, * d_c;
    cudaStream_t stream[20];

    dim3 blk(32, 32, 1);
    dim3 grid((M + blk.x - 1) / blk.x, (N + blk.y - 1) / blk.y, 1);

    int size_a = X * N * K;
    int size_b = K * M;
    int size_c = X * N * M;

    cudaMalloc((void**)&d_a, size_a * sizeof(float));
    cudaMalloc((void**)&d_b, size_b * sizeof(float));
    cudaMalloc((void**)&d_c, size_c * sizeof(float));

    int segmentSize = N * K;

    for (int i = 0; i < X; ++i)
    {
        cudaStreamCreate(&stream[i]);
    }

    for (int i = 0; i < X; ++i)
    {
        int offset = i * segmentSize;
        cudaMemcpyAsync(&d_a[offset], &A[offset], segmentSize * sizeof(float), cudaMemcpyHostToDevice, stream[i]);
        cudaMemcpyAsync(&d_b[offset], &B[offset], M * K * sizeof(float), cudaMemcpyHostToDevice, stream[i]);
        kernel_multiply << <grid, blk, 0, stream[i] >> > (d_a + offset, d_b, d_c + i*MxN, M, N, K);
        cudaMemcpyAsync(&C[offset], &d_c[offset], N * M * sizeof(float), cudaMemcpyDeviceToHost, stream[i]);

    }

    for (int i = 0; i < X; ++i)
    {
        cudaStreamSynchronize(stream[i]);
        cudaStreamDestroy(stream[i]);
    }
}
```

Host function

```
def main(task='mrpc',
         train_cfg='config/train_cola.json',
         model_cfg='config/bert_base.json',
         data_file='data.tsv',
         model_file=None,
         pretrain_file='weight.pt',
         data_parallel=False,
         vocab="C:/Users/prbhatnagar/Downloads/Final_Project/py_src/uncased_L-12_H-768_A-12/bert-base-uncased-vocab.txt",
         save_dir='mrpc',
         max_len=128,
         mode='eval'):

    cfg = train.Config.from_json(train_cfg)
    model_cfg = models.Config.from_json(model_cfg)

    set_seeds(cfg.seed)

    tokenizer = tokenization.FullTokenizer(vocab_file=vocab, do_lower_case=True)
    TaskDataset = dataset_class(task)

    print('Dataset and Tokenizer loaded successfully')
    print()

    pipeline = [Tokenizing(tokenizer.convert_to_unicode, tokenizer.tokenize),
                AddSpecialTokensWithTruncation(max_len),
                TokenIndexing(tokenizer.convert_tokens_to_ids,
                              TaskDataset.labels, max_len)]
    dataset = TaskDataset(data_file, pipeline)
    data_iter = DataLoader(dataset, batch_size=cfg.batch_size, shuffle=True)

    print('Pipeline implemented successfully')
    print()
```

Entry point - Tokenizer, Dataset, pipeline initialized

```
    model = Classifier(model_cfg, len(TaskDataset.labels))
    criterion = nn.CrossEntropyLoss()
    print('Model created successfully')
    print()

    trainer = train.Trainer(cfg,
                            model,
                            data_iter,
                            optim.optim4GPU(cfg, model),
                            save_dir, get_device())

    state_dict = torch.load(pretrain_file, map_location =torch.device('cpu'))
    trainer.model.load_state_dict(state_dict, strict = False)
    print('Trainer created and weights loaded successfully')
    print()
```

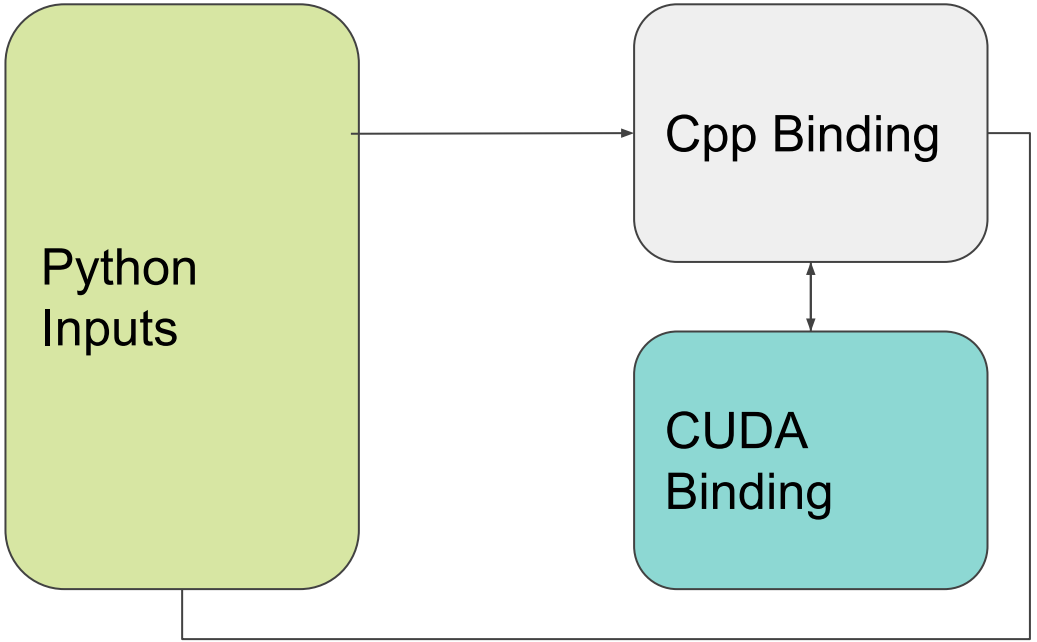
Model created & weights loaded

```
if mode == 'eval':
    def evaluate(model, batch):
        input_ids, segment_ids, input_mask, label_id = batch
        print(' Starting inference')
        print()
        print('BERT says HI!')
        print()

        initial_time_seconds = time.time()
        initial_time_milliseconds = int(initial_time_seconds * 1000)

        logits = model(input_ids, segment_ids, input_mask)

        final_time_seconds = time.time()
        final_time_milliseconds = int(final_time_seconds * 1000)
        print()

        print(f"Total Inference time (GPU) in Milliseconds: {final_time_milliseconds - initial_time_milliseconds}")

        _, label_pred = logits.max(1)
        result = (label_pred == label_id).float()
        accuracy = result.mean()
        return accuracy, result

    results = trainer.eval(evaluate, model_file, data_parallel)
```

Model Inference

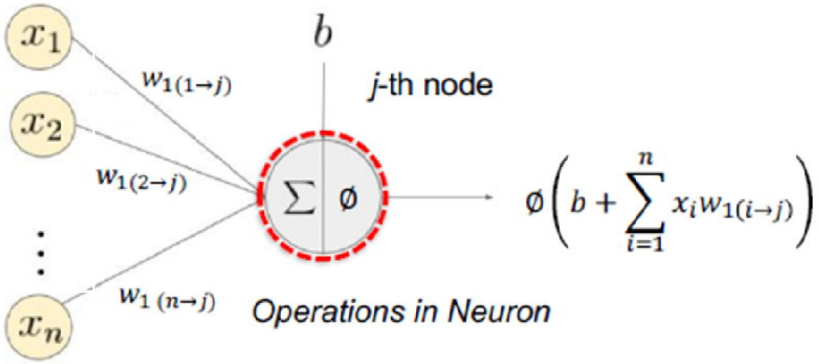
Pybind11: C++ Framework

```
PYBIND11_MODULE(cu_multiply_matrix, m) {
    m.def("multiply", &mm_wrapper, "Linear layer");

#ifdef VERSION_INFO
    m.attr("__version__") = VERSION_INFO;
#else
    m.attr("__version__") = "dev";
#endif
}
```



Fully Connected Layer

- The linear layer has a form of:
  Y = X . W
- The input is an image which is can be represented as a linear array, ie., by making the 2D array into a linear array
- Thus, the operation in the neuron can now be represented by matrix multiplication.
- Here, custom CUDA matrix multiplication function is used to model it.

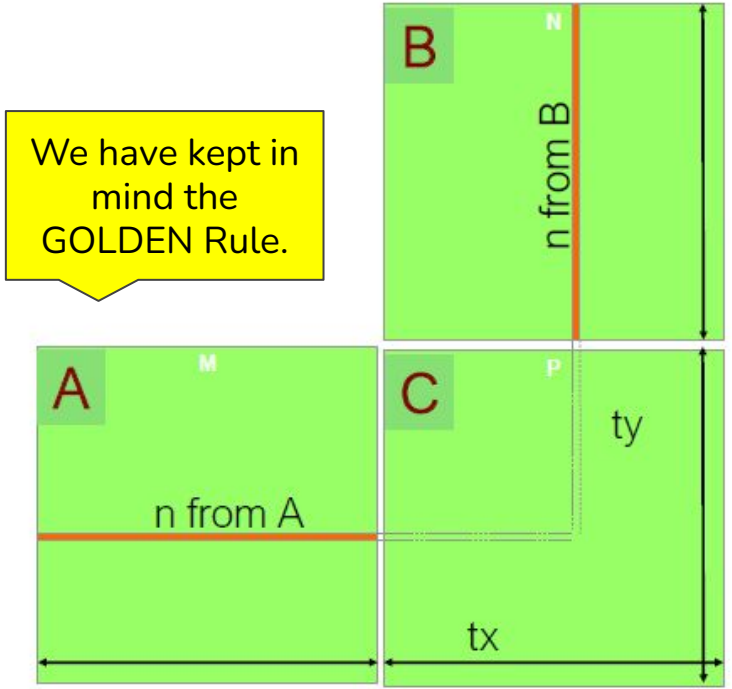


$$\begin{bmatrix} w_{1(1\to1)} & \cdots & w_{1(5\to1)} \\ \vdots & \ddots & \vdots \\ w_{1(1\to N)} & \cdots & w_{1(5\to N)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_{21} \\ y_{22} \\ y_{23} \\ \cdots \\ y_{2N} \end{bmatrix}$$

Matrix multiplication
(bias omitted for simplicity)

Matrix Multiplication in CUDA

- Each thread corresponds to an element in C.
- This calls a row from matrix A and a column from matrix B.
- The block size used here is (32, 32)
- Keeping the block dimensions a multiple of 32 results in memory coalescing.
- This increases the performance for the chosen block size.
- Since the size of the matrix is not fixed from the start, the grid size is calculated by the host function



Multistream Processing for 3D Matrix Multiplication

- The input is a sentence which is divided into 128 (*N*) tokens and every token has an embedding size of 768 (*K*).
- In multi-head attention module, this input is matrix multiplied with Projection Q, K, V matrices
- To process multiple sentences at once, we use CUDA multi-stream processing to implement a 3D matrix multiplication:

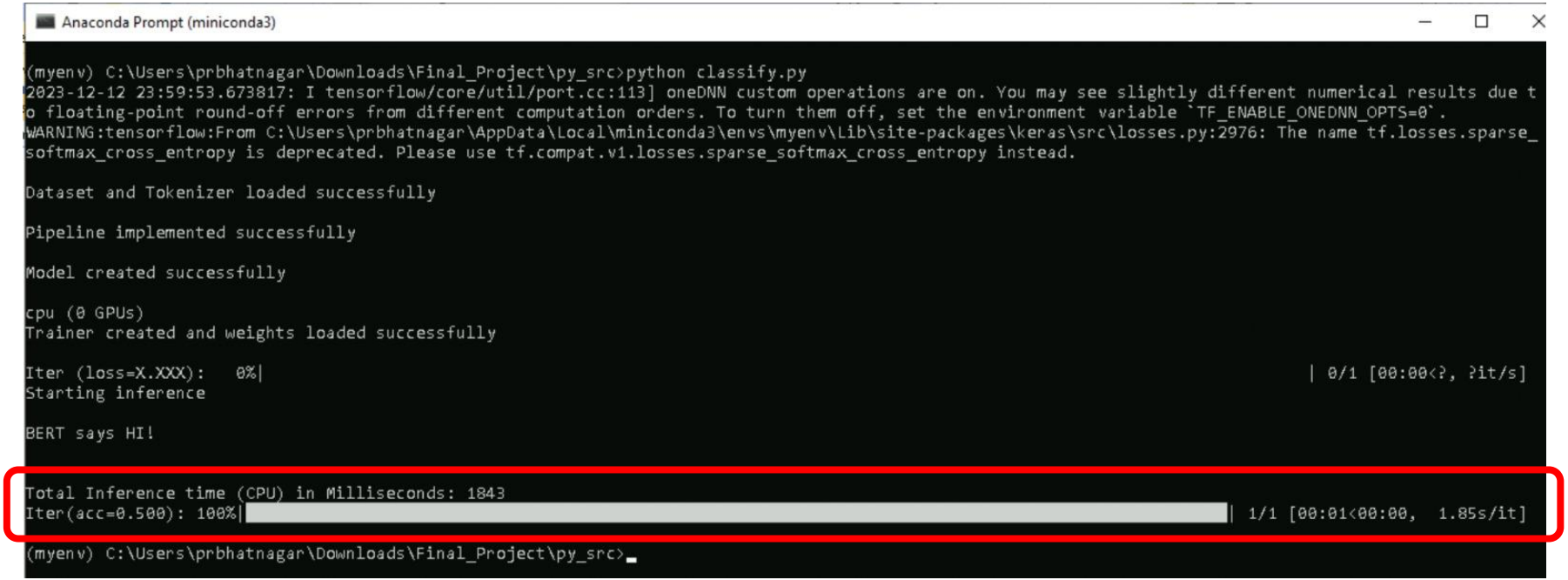
  Matrix A: X x N x K, Matrix B: K x M

  Here,
  X = batch size = #sentences processed together
- Includes asynchronous memory operations with CUDA streams for simultaneous data transfer and computation.

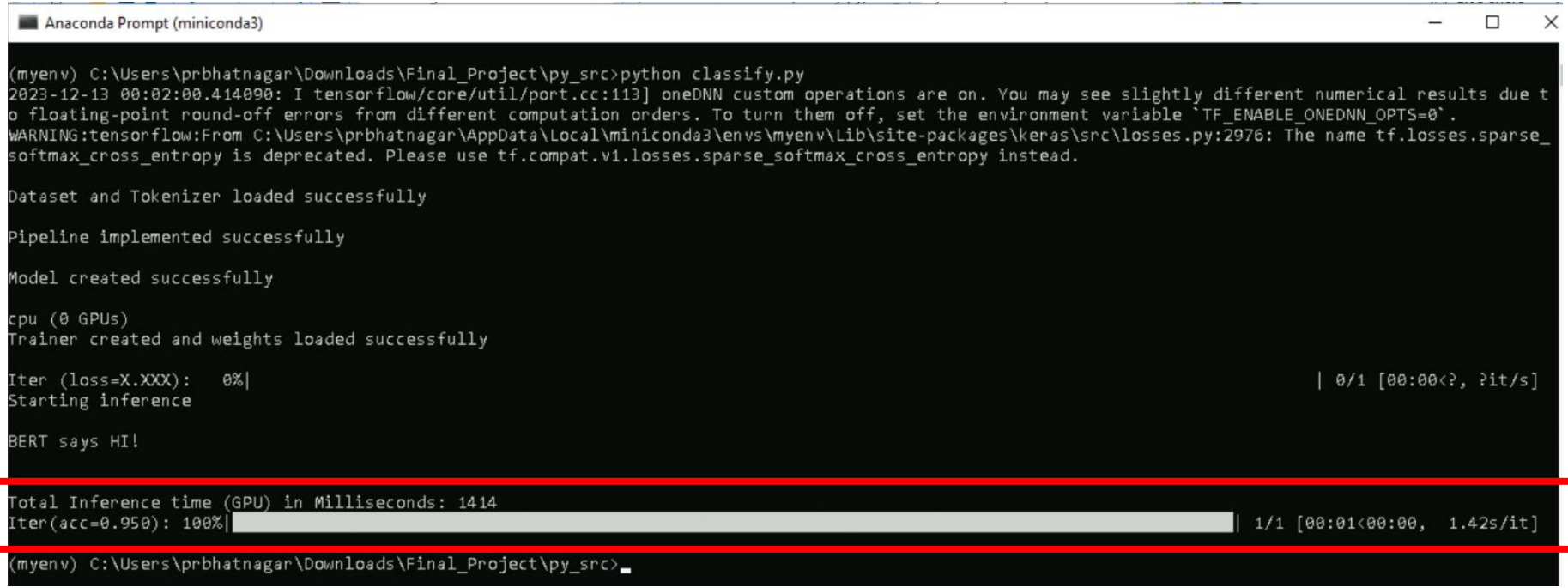
```
void cu_multiply(float* A, float* B, float* C, int M, int N, int K, int X)
{
    // Assume A = XxNxK, B = KxM matrix, C = XxNxM
    float* d_a, * d_b, * d_c;
    cudaStream_t stream[20];

    dim3 blk(32, 32, 1);
    dim3 grid((M + blk.x - 1) / blk.x, (N + blk.y - 1) / blk.y, 1);

    int size_a = X * N * K;
    int size_b = K * M;
    int size_c = X * N * M;

    cudaMalloc((void**)&d_a, size_a * sizeof(float));
    cudaMalloc((void**)&d_b, size_b * sizeof(float));
    cudaMalloc((void**)&d_c, size_c * sizeof(float));

    int segmentSize = N * K;

    for (int i = 0; i < X; ++i)
    {
        cudaStreamCreate(&stream[i]);
    }

    for (int i = 0; i < X; ++i)
    {
        int offset = i * segmentSize;
        cudaMemcpyAsync(&d_a[offset], &A[offset], segmentSize * sizeof(float), cudaMemcpyHostToDevice, stream[i]);
        cudaMemcpyAsync(&d_b[offset], &B[offset], M * K * sizeof(float), cudaMemcpyHostToDevice, stream[i]);
        kernel_multiply << <grid, blk, 0, stream[i] >> > (d_a + offset, d_b, d_c + i*M*N, M, N, K);
        cudaMemcpyAsync(&C[offset], &d_c[offset], N * M * sizeof(float), cudaMemcpyDeviceToHost, stream[i]);

    }

    for (int i = 0; i < X; ++i)
    {
        cudaStreamSynchronize(stream[i]);
        cudaStreamDestroy(stream[i]);
    }
```

Results

CPU Inference Time ~ 20 samples



GPU Inference Time ~ 20 samples



References

- Attention is All You Need
- https://huggingface.co/bert-base-uncased
- https://pytorch.org/docs/stable/index.html
- https://pybind11.readthedocs.io/en/stable/basics.html
- https://github.com/MegEngine/Models/blob/master/official/nlp/bert/glue_data/MRPC

Thank you