

멀티코어프로그래밍: 프로젝트

Transformer 모델 병렬처리 – 코드 설명

세종대학교 컴퓨터공학과
박기호

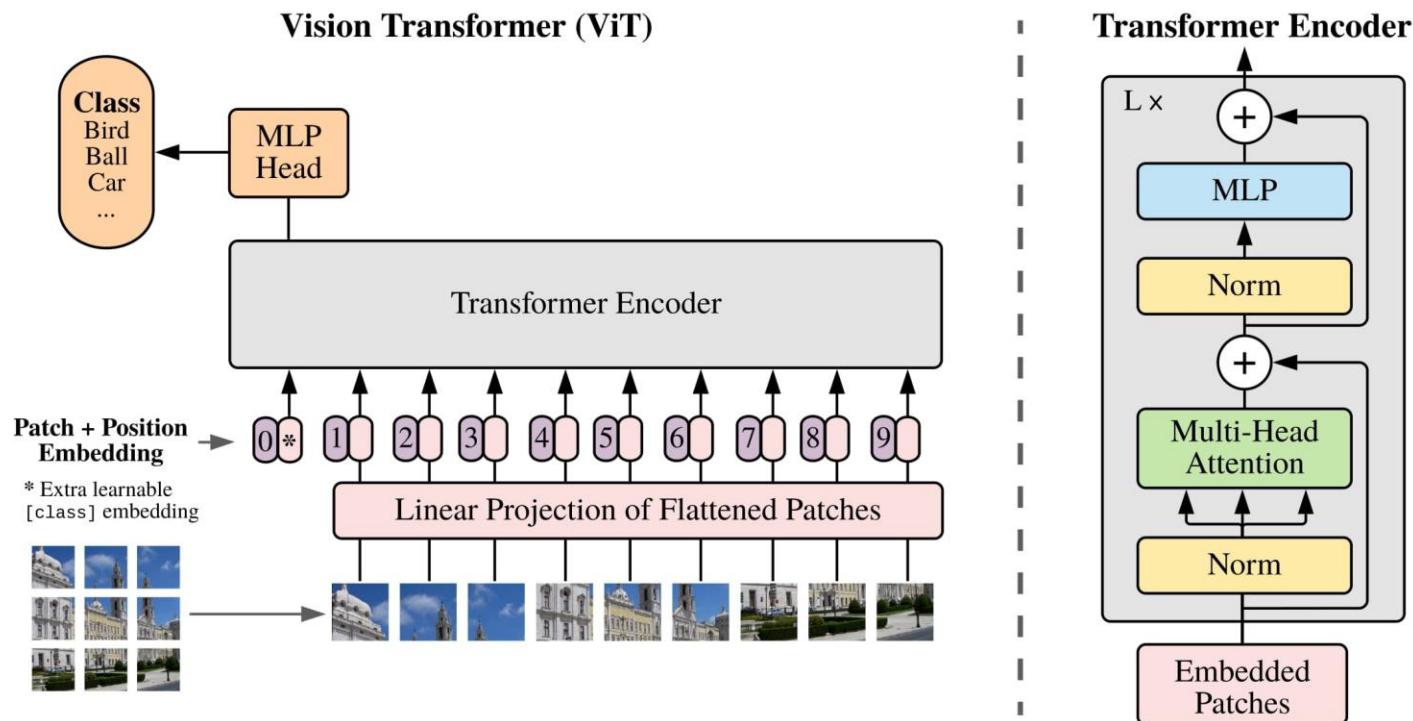
프로젝트 개요

- 인공신경망을 활용하여 이미지를 분류하는 예제
 - 사전 학습된 모델(ViT16)을 사용해 이미지 분류를 실행
- Vision Transformer를 활용해 ImageNet-1000 데이터셋에 대한 추론 수행
- OpenCL을 사용하여 Transformer 모델 GPU 가속 구현
 - 정답과 동일한 결과를 출력
 - 수행 시간이 짧을 수록 높은 점수 부여 (이미지 데이터 100장 기준)
 - + 수행 시간 뿐 아니라 적용한 가속 기법도 확인

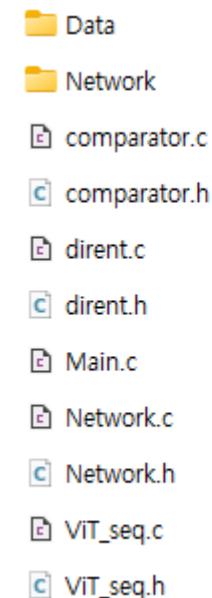
Vision Transformer

- **Vision Transformer**

- Transformer의 인코더 구조를 활용하여 이미지를 분류하는 모델
- 이미지를 고정 크기의 패치로 나누고, 각 패치를 임베딩하여 시퀀스 형태로 Transformer에 입력



폴더 구조

- **./Data**
 - Input-100.bin : ImageNet data 100장 (test용 data)
 - answer_result.txt : input data의 정답 label 및 probability
 - opencl_result.txt : 구현한 모델의 출력값 (answer_result와 비교를 통해 검증)
 - **./Network**
 - Vision Transformer에서 사용되는 152개의 weights 파일
 - **Code File**
 - Main.c : Main 함수
 - Network.c : 입력 데이터 및 가중치 관리용 파일
 - Comparator.c : 정답 라벨과 출력 라벨의 비교
 - DIRENT.c : 폴더 관리(open, read)용 파일
 - ViT_seq.c : Vision Transformer의 sequential한 code
 - + Opencl 파일과 커널 파일을 생성하여 프로젝트 진행
- 

Main.c

- 초기 세팅

```
16
17 int main() {
18     ////////////////////////////////////////////////////////////////// Input load //////////////////////////////////////////////////////////////////
19     const char* img_filename = "./Data/input-100.bin";
20     ImageData* images = load_image_data(img_filename);
21     if (images == NULL) return 1;
22
23     int image_size = images->c * images->h * images->w;
24
25     ////////////////////////////////////////////////////////////////// Weight load //////////////////////////////////////////////////////////////////
26     Network network[152];
27     load_weights("./Network", network, 152);
28
29     ////////////////////////////////////////////////////////////////// Model //////////////////////////////////////////////////////////////////
30     int n = images->n;
31     float** probabilities = (float**)malloc(sizeof(float*) * n);
32     for (int i = 0; i < n; i++) {
33         probabilities[i] = (float*)malloc(sizeof(float) * 1000);
34     }
35
36     time_t start, end;
37
38     FILE* fp_output = NULL;
39     errno_t err = fopen_s(&fp_output, "./Data/opencl_result.txt", "w");
40     if (err != NULL) {
41         printf("Error: cannot open ./Data/opencl_result.txt for writing\n");
42         return 1;
43     }
```

Image Data load

Weights load

결과(probabilities)를 저장할
float 배열 생성

Main.c

- 모델 실행

```
printf("=====Start=====#\n");
start = clock();
// Input here
ViT_seq(images, network, probabilities);
end = clock();
printf("Elapsed time: %.2f sec#\n", (double)(end - start) / CLK_TCK);

int pred_idx = 0;
for (int i = 0; i < n; i++) {
    for (int j = 1; j < 1000; j++) {
        float cur = probabilities[i][j];
        if (cur > probabilities[i][pred_idx]) {
            pred_idx = j;
        }
    }
    const char* pred_label = imagenet_label[pred_idx];
    fprintf(fp_output, "%d's image label : %s (index: %d, probability : %f)\n";
}
fclose(fp_output);
```

ViT 모델 실행

실행시간 출력

결과값 파일에 저장

Main.c

- Comparator 호출

```
int pred_idx = 0;
for (int i = 0; i < n; i++) {
    for (int j = 1; j < 1000; j++) {
        float cur = probabilities2[i][j];
        if (cur > probabilities2[i][pred_idx]) {
            pred_idx = j;
        }
    }
    const char* pred_label = imagenet_label[pred_idx];
    fprintf(fp_output, "[%d] label: %d / prob: %.6f\n", i, pred_idx, probabilities2[i][pred_idx]);
}
fclose(fp_output);

comparator();
```

- Comparator()를 호출하여, 정답 라벨과 출력 라벨의 값을 비교
 - + 자세한 내용은 comparator.c 참고

ViT_seq.c

• Encoder 함수

```
////////// Encoder Architecture //////////
void Encoder(float* input, float* output,
    Network ln1_w, Network ln1_b, Network attn_w, Network attn_b, Network attn_out_w,
    Network ln2_w, Network ln2_b, Network mlp1_w, Network mlp1_b, Network mlp2_w,
    int tokens = ((img_size / patch_size) * (img_size / patch_size)) + 1;
    float* ln1_out = (float*)malloc(sizeof(float) * tokens * embed_dim);
    float* attn_out = (float*)malloc(sizeof(float) * tokens * embed_dim);
    float* residual = (float*)malloc(sizeof(float) * tokens * embed_dim);
    float* ln2_out = (float*)malloc(sizeof(float) * tokens * embed_dim);
    float* mlp_out = (float*)malloc(sizeof(float) * tokens * embed_dim);

/*LN1*/
layer_norm(input, ln1_out, ln1_w, ln1_b);

/*Attn*/
multihead_attn(ln1_out, attn_out, attn_w, attn_b, attn_out_w, attn_out_b);

/*Residual1*/
for (int i = 0; i < tokens * embed_dim; i++) {
    residual[i] = input[i] + attn_out[i];
}

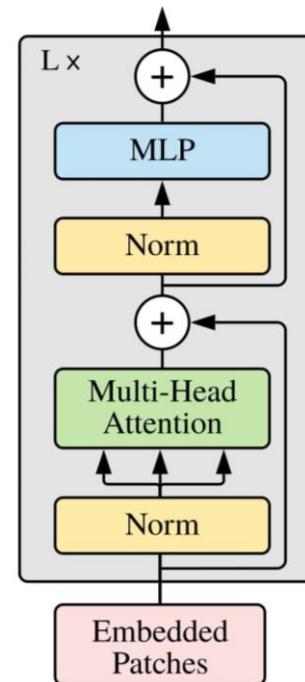
/*LN2*/
layer_norm(residual, ln2_out, ln2_w, ln2_b);

/*MLP*/
mlp_block(ln2_out, mlp_out, mlp1_w, mlp1_b, mlp2_w, mlp2_b);

/*Residual2*/
for (int i = 0; i < tokens * embed_dim; i++) {
    output[i] = residual[i] + mlp_out[i];
}

free(ln1_out); free(attn_out); free(residual); free(ln2_out); free(mlp_out)
```

Vision Transformer의 Encoder와
동일하게 구현

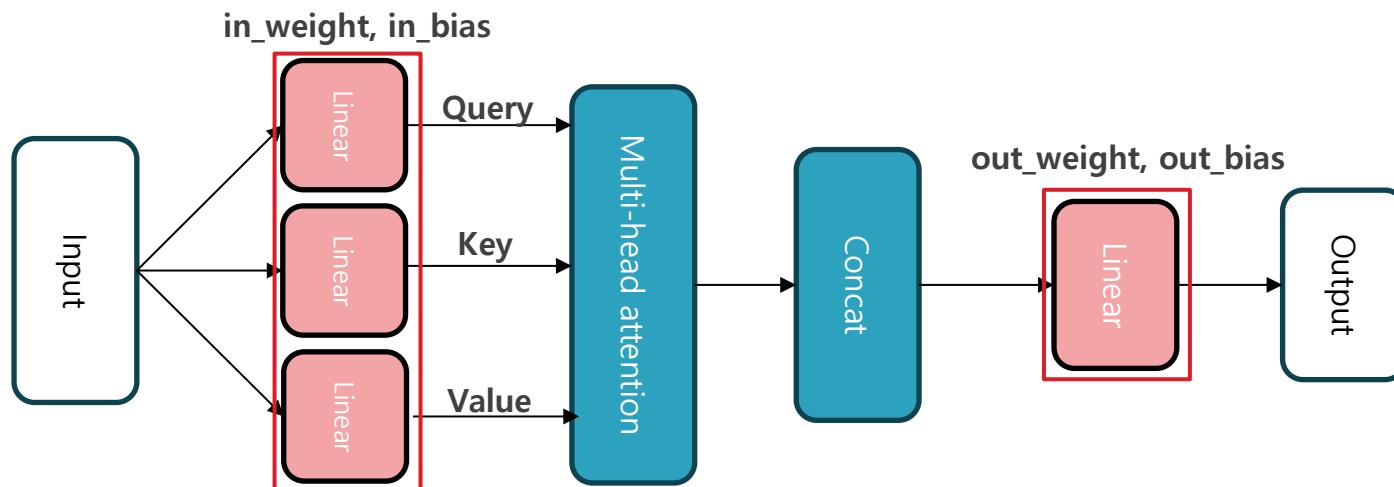


ViT_seq.c

- Multi-Head Attention

```
void multihead_attn(float* input, float* output,  
    Network in_weight, Network in_bias, Network out_weight, Network out_bias) {
```

- in_weight, in_bias : Query, Key, Value 생성을 위한 network
 - + 하나의 network 안에 3개에 대한 weight와 bias가 들어있음
 - + Network size = (token * embed_dim) * 3
- out_weight, out_bias : final linear projection을 위한 network
 - + Network size = (token * embed_dim)



ViT_seq.c

- Multi-Head Attention : ① Query, key, Value 생성

```
/*Allocate Q, K, V : tokens * dim*/
int Q_dim = 0, K_dim = embed_dim, V_dim = embed_dim * 2;
float* Q = (float*)malloc(sizeof(float) * tokens * embed_dim);
float* K = (float*)malloc(sizeof(float) * tokens * embed_dim);
float* V = (float*)malloc(sizeof(float) * tokens * embed_dim);

/*Q, K, V 구하기*/
for (int t = 0; t < tokens; t++) {
    float sum_q, sum_k, sum_v;
    for (int i = 0; i < embed_dim; i++) {
        sum_q = in_bias.data[Q_dim + i], sum_k = in_bias.data[K_dim + i], sum_v = in_bias.data[V_dim + i];
        for (int j = 0; j < embed_dim; j++) {
            sum_q += input[t * embed_dim + j] * in_weight.data[(Q_dim + i) * embed_dim + j];
            sum_k += input[t * embed_dim + j] * in_weight.data[(K_dim + i) * embed_dim + j];
            sum_v += input[t * embed_dim + j] * in_weight.data[(V_dim + i) * embed_dim + j];
        }
        Q[t * embed_dim + i] = sum_q;
        K[t * embed_dim + i] = sum_k;
        V[t * embed_dim + i] = sum_v;
    }
}

int print_tokens = tokens < 5 ? tokens : 5;
int print_dims = embed_dim < 10 ? embed_dim : 10;

/*Attn 결과를 저장할 버퍼*/
float* attn_output = (float*)malloc(sizeof(float) * tokens * embed_dim);
for (int i = 0; i < tokens * embed_dim; i++) attn_output[i] = 0.0f;
```

Query = (Input * weight_q) + bias_q
Key = (Input * weight_k) + bias_k
Value = (Input * weight_v) + bias_v

Query, Key, Value 생성

ViT_seq.c

- Multi-Head Attention : ② Scaled-dot Attention

```

/* head별로 attn 수행 */
for (int h = 0; h < num_heads; h++) {
    int head_offset = h * head_dim;

    // attn_score 저장 공간
    float* scores = (float*)malloc(sizeof(float) * tokens * tokens);
    float* scores_tmp = (float*)malloc(sizeof(float) * tokens * tokens);

    // 각 head에 대해 scaled-dot attn
    for (int i = 0; i < tokens; i++) {
        for (int j = 0; j < tokens; j++) {    Query와 Key의 matmul
            float score = 0.0f;
            for (int d = 0; d < head_dim; d++) {
                float q = Q[i * embed_dim + head_offset + d];
                float k = K[j * embed_dim + head_offset + d];
                score += q * k;
            }
            scores[i * tokens + j] = score / sqrtf((float)head_dim);
        }
    }
}

```

Header별로 수행

```

// softmax 적용
for (int i = 0; i < tokens; i++) {
    float max_val = scores[i * tokens];
    for (int j = 1; j < tokens; j++) {
        if (scores[i * tokens + j] > max_val) max_val = scores[i * tokens + j];
    }
    float sum_exp = 0.0f;
    for (int j = 0; j < tokens; j++) {
        scores[i * tokens + j] = expf(scores[i * tokens + j] - max_val);
        sum_exp += scores[i * tokens + j];
    }
    for (int j = 0; j < tokens; j++) {
        scores[i * tokens + j] /= sum_exp;
    }
}

```

Softmax 함수

```

// scores와 V를 곱해 head output 계산
float* head_out = (float*)malloc(sizeof(float) * tokens * head_dim);
for (int i = 0; i < tokens; i++) {
    for (int d = 0; d < head_dim; d++) {    Value와 matmul
        float sum = 0.0f;
        for (int j = 0; j < tokens; j++) {
            sum += scores[i * tokens + j] * V[j * embed_dim + head_dim + d];
        }
        head_out[i * head_dim + d] = sum;
    }
}

```

Concat

```

// head_out을 attn_output의 해당 부분에 복사
for (int i = 0; i < tokens; i++) {
    for (int d = 0; d < head_dim; d++) {
        attn_output[i * embed_dim + head_offset + d] = head_out[i * head_dim + d];
    }
}

```

```

free(scores);
free(head_out);

```

```

free(Q); free(K); free(V);

```

ViT_seq.c

- Multi-Head Attention : ③ Final linear Projection

```
// 최종 선형 프로젝션
for (int t = 0; t < tokens; t++) {
    for (int i = 0; i < embed_dim; i++) {
        float sum = out_bias.data[i];
        for (int j = 0; j < embed_dim; j++) {
            sum += attn_output[t * embed_dim + j] * out_weight.data[i * embed_dim + j];
        }
        output[t * embed_dim + i] = sum;
    }
}
free(attn_output);
```

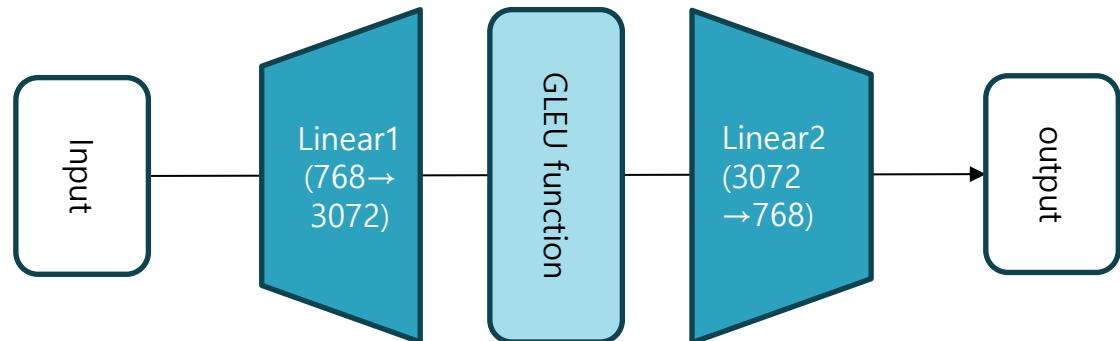
- 마지막 연산까지 진행 후, Multi-head Attention의 output 출력

ViT_seq.c

- Multi-Layer Perceptron

```
void mlp_block(float* input, float* output,
    Network fc1_weight, Network fc1_bias, Network fc2_weight, Network fc2_bias) {
    int tokens = ((img_size / patch_size) * (img_size / patch_size)) + 1; //197
    int Embed_dim = embed_dim; //768
    int hidden_dim = ((int)(embed_dim * mlp_ratio)); //3072
```

- fc1_weight, fc1_bias : 임베딩 차원을 더 큰 차원으로 확장하는 첫번째 Linear의 network
 - + Network size = (token * hidden_dim)
- fc2_weight, fc2_bias : 원래의 임베딩 차원으로 축소하는 두번째 Linear의 network
 - + Network size = (token * embed_dim)



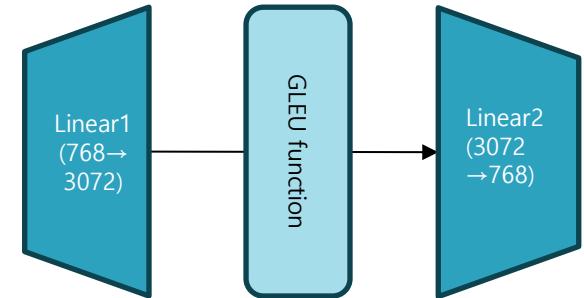
ViT_seq.c

- Multi-Layer Perceptron

```
void mlp_block(float* input, float* output,
    Network fc1_weight, Network fc1_bias, Network fc2_weight, Network fc2_bias) {

    int tokens = ((img_size / patch_size) * (img_size / patch_size)) + 1; //197
    int Embed_dim = embed_dim; //768
    int hidden_dim = ((int)(embed_dim * mlp_ratio)); //3072
    float* fc1_out = (float*)malloc(sizeof(float) * tokens * hidden_dim);

    linear_layer(input, fc1_out, tokens, embed_dim, hidden_dim, fc1_weight, fc1_bias);
    // GELU 활성화
    for (int i = 0; i < tokens * hidden_dim; i++) {
        fc1_out[i] = gelu(fc1_out[i]); // activation function
    }
    linear_layer(fc1_out, output, tokens, hidden_dim, embed_dim, fc2_weight, fc2_bias);
    free(fc1_out);
}
```



```
void linear_layer(float* input, float* output, int tokens, int in_features, int out_features,
    Network weight, Network bias) {
    for (int t = 0; t < tokens; t++) {
        for (int o = 0; o < out_features; o++) {
            float sum = bias.data[o];
            for (int i = 0; i < in_features; i++) {
                sum += input[t * in_features + i] * weight.data[o * in_features + i];
            }
            output[t * out_features + o] = sum;
        }
    }
}
```

Linear용 function