110062118
賴姿妘

HW6 OpenAcc

(1) 優化過程

在這份作業中，我首先使用 openAcc 進行 for loop 等級的平行化，產生 version 1。接著，透過 create memory 減少 memory copy 所需時間以及嘗試使用 async 以及 wait。最後，繼續針對各個函式進行優化。

(2) 函式優化

✧ Padding2D

i. Version 1

使用簡單的#pragama parallel loop 進行平行化。

```
void Padding2D(float *A, float *B, int n, int size) {
    int p = (K - 1) / 2;

    #pragma acc parallel loop copyin(A[0:n*size*size]) copyout(B[0:n*(size+2*p)*(size+2*p)])
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < size + 2 * p; j++) {
            for (int k = 0; k < size + 2 * p; k++) {
                if (j < p || j >= size + p || k < p || k >= size + p) {
                    B[i * (size + 2 * p) * (size + 2 * p) + j * (size + 2 * p) + k] = 0;
                } else {
                    B[i * (size + 2 * p) * (size + 2 * p) + j * (size + 2 * p) + k] = A[i * size * size + (j - p) * size + (k - p)];
                }
            }
        }
    }
}
```

ii. Version 2

使用變數將重複計算的值例如 padded_size 儲存，減少計算所花時間。在原始程式碼中，每次都要判斷是否在矩陣範圍內，導致 divergence 發生，以及大量不必要的條件判斷。在此版本，我將上下左右邊框的 padding 填充獨立在兩個 loop 處理，便可減少條件判斷的次數。

平行化方面，使用#pragma acc loop vector 將迴圈任務轉乘 SIMD 指令。

```
int p = (K - 1) / 2;
int padded_size = size + 2 * p;
int size_size = size * size;
int padded_size_padded_size = padded_size * padded_size;

#pragma acc parallel loop present(A[0:n * size_size], B[0
for (int i = 0; i < n; i++) {
    float *A_ptr = A + i * size_size;
    float *B_ptr = B + i * padded_size_padded_size;
```

110062118
賴姿妘

```
// Fill top and bottom edges
#pragma acc loop vector
for (int j = 0; j < p; j++) {
    #pragma acc loop vector
    for (int k = 0; k < padded_size; k++) {
        B_ptr[j * padded_size + k] = 0;
        B_ptr[(padded_size - 1 - j) * padded_size + k] = 0;
    }
}

// Fill left and right edges and copy inner elements
#pragma acc loop vector
for (int j = p; j < size + p; j++) {
    int j_padded_size = j * padded_size;
    #pragma acc loop vector
    for (int k = 0; k < p; k++) {
        B_ptr[j_padded_size + k] = 0;
        B_ptr[j_padded_size + padded_size - 1 - k] = 0;
    }
    #pragma acc loop vector
    for (int k = p; k < size + p; k++) {
        B_ptr[j_padded_size + k] = A_ptr[(j - p) * size + (k - p)];
    }
}
```

✧ Conv2D

i. Version 1

使用#pragma acc parallel gang vector 進行平行化。使用 collapse 摺疊迴圈，提高並行性,更好地利用 GPU 的資源。利用 reduction 修飾符產生捲機結果。

```
#pragma acc parallel loop collapse(2) gang vector
for (int i = 0; i < n; i++) {
    for (int j = 0; j < cout; j++) {
        #pragma acc loop collapse(2)
        for (int x = 0; x < size; x++) {
            for (int y = 0; y < size; y++) {
                int d_index = i * cout * size * size + j * size * size + x * size + y;
                D[d_index] = C[j];

                #pragma acc loop collapse(3) reduction(+:D[d_index])
                for (int k = 0; k < cin; k++) {
```

ii. Version2

使用 local sum 將處理 output 值計算，最後才將計算後的值更新到 global memory 上的值。

110062118
賴姿妘

```
float sum = C[j];

#pragma acc loop collapse(3) reduction(+:sum)
for (int k = 0; k < cin; k++) {
    for (int kx = 0; kx < K; kx++) {
        for (int ky = 0; ky < K; ky++) {
```

✧ ReLU

使用#pragma acc parallel 進行平行化。針對 float 型別的 max 操作使用 fmax 以達到更快的速度。

```
void ReLU(float *A, int n) {
    #pragma acc parallel loop copyin(A[0:n]) copyout(A[0:n])
    for (int i = 0; i < n; i++) {
        A[i] = fmaxf(0.0f, A[i]);
    }
}
```

✧ MaxPool2D

使用 collapse, gang, vector 進行優化。使用 idx 儲存運算結果，節省運算時間。

```
#pragma acc parallel loop collapse(3) gang vector present(A[0:n * size * s
for (int i = 0; i < n; i++) {
    for (int j = 0; j < pool_size; j++) {
        for (int k = 0; k < pool_size; k++) {
            int idx = i * size * size + (2 * j) * size + (2 * k);
            float max_val = A[idx];

            max_val = fmaxf(max_val, A[idx + 1]);
            max_val = fmaxf(max_val, A[idx + size]);
            max_val = fmaxf(max_val, A[idx + size + 1]);

            B[i * pool_size * pool_size + j * pool_size + k] = max_val;
        }
    }
}
```

✧ LinearLayer

起初使用 acc loop 進行優化，發現加速並不明顯，但 linearlayer 應該佔多數運算時間，經過搜尋後，發現可能是編譯器認為迴圈有相依性，便無進行平行化，因此加上 independent，讓編譯器可以進行優化。使用 reduction 將每個 thread 計算結果相加。

110062118
賴姿妘

```
void LinearLayer(float *A, float *B, float *C, float *D, int n, int k, int m) {
    #pragma acc kernels copyin(A[0:n*k], B[0:k*m], C[0:m]) copyout(D[0:n*m])
    {
        #pragma acc loop independent
        for (int i = 0; i < n; i++) {
            #pragma acc loop independent
            for (int j = 0; j < m; j++) {
                float sum = C[j];
                #pragma acc loop reduction(+:sum)
                for (int a = 0; a < k; a++) {
                    sum += A[i * k + a] * B[a * m + j];
                }
                D[i * m + j] = sum;
            }
        }
    }
}
```

(3) 減少資料搬運時間

若單純優化函式，layer 和 layer 之間都要進行資料的搬移，造成時間的浪費。

在執行時，在 GPU 上 allocate 儲存中間結果的 layer，便不用在兩個裝置間不停搬移。

```
#pragma acc data create(co
#pragma acc data copyin(tr
#pragma acc data copyin(co
#pragma acc data copyin(co
#pragma acc data copyin(fc
#pragma acc data copyin(fc
{
    ConvolutionLayer(train
    ConvolutionLayer(conv1
    LinearLayer(conv2_outp
    LinearLayer(fc1_output
    Argmax(fc2_output, res
}

delete[] conv1_output;
delete[] conv2_output;
delete[] fc1_output;
delete[] fc2_output;
```

110062118
賴姿妘

嘗試使用 async 和 wait，來 overlap 運算和資料搬移時間，使用後發現反而速度較慢。

```
#pragma acc data create(conv1
#pragma acc data copyin(train
#pragma acc data copyin(conv1

#pragma acc data copyin(conv2

#pragma acc data copyin(fc1_w
#pragma acc data copyin(fc2_w
{
    #pragma acc wait(1)
    ConvolutionLayer(training
                     num_image

    #pragma acc wait(1, 2)
    ConvolutionLayer(conv1_ou
                     num_image

    #pragma acc wait(2, 3)
    LinearLayer(conv2_output,
                num_images, F

    #pragma acc wait(3, 4)
    LinearLayer(fc1_output, f
                num_images, F

    #pragma acc wait(4)
    Argmax(fc2_output, result

}
```

110062118
賴姿妘

(4) Ablation study

openACC	Memory movement optimization	Version2 Padding2D	Version2 Conv2D	async	Time(ms)
					201267 (sequential)
★					20836
★	★				18704
★	★	★			18622
★	★		★		422
★	★	★		★	464
★	★	★	★		351

(5) Pros and Cons

✧ Pros

OpenACC 十分簡單好上手，提供程式碼的可維護性，也很方便擴展到不同 GPU 上

✧ Cons

由於將平行化交由編譯器處理，以此若沒有特別查看，開發者無法明確了解編譯結果。若程式碼過於複雜，過於簡單的指令可能使編譯器跳過可平行化的部分。另外與手動優化相比，由於 OpenAcc 讓使用者忽略了底層細節，可能沒辦法做到極致的速度優化。

(6) What challenges did you face when parallelizing the network?

由於每個 input 的維度很高，因此 layer 中的 function 有超級多層，在 trace code 的時候要花點時間去理解目前的層級，而且由於都是用一維陣列表示，在 index 處理上要有點小心，例如在更改 padding 2d 的時候。另外，也花了不少時間測試不同裝飾符的效果，例如 vector collapse 等。還有，有的時候以為用較粗略的指令編譯器就可以進行優化，後來才發現要加 independent，才能順利優化。