高階合成技術於應用加速

Application Acceleration with High-Level Synthesis

Lab #C Vitis Library

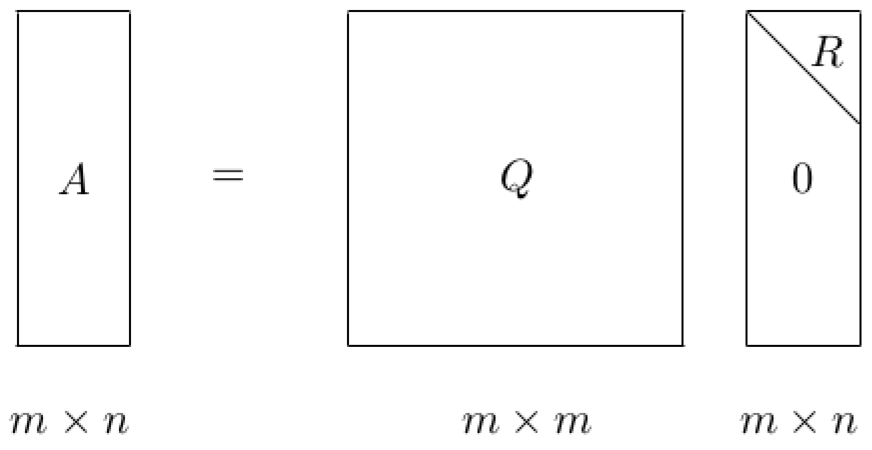
QR Factorization

110061608 吳承哲

110061592 許睿哲

• Background introduction

QR Factorization：QR分解是把矩陣分解成一個正交矩陣和一個上三角矩陣，經常用來解least squares method。QR分解也是求特徵值有效且廣泛應用的方法。下圖為分解示意圖：



其中，Q 是一個標準正交矩陣，R 是上三角矩陣。

QR分解常見的方法有三種：

Gram-Schmidt orthogonalization,

Householder transformation,

Givens rotation.

這個example code使用的是 Householder transformation。

Householder transformation 是將一個向量關於某個平面或者超平面進行反射。我們可以利用這個操作對m * n(m ≥ n)的矩陣A進行QR分解。令 x 為矩陣 A 的任一 m 維實列向量且$||x|| = |\alpha|$，e_1為單位向量$(1, 0, \dots, 0)^T$

$A = QR$

$v = x - \alpha e_1$

$H = I - \beta v v^T$, $\beta = \frac{2}{\langle v,v \rangle}$

當矩陣H乘上一次矩陣A，會得到矩陣R的一列向量，

舉例來說，如果要對A做QR分解：

$$A = \begin{bmatrix} 0 & 3 & 1 \\ 0 & 4 & -2 \\ 2 & 1 & 1 \end{bmatrix}$$ ，$x_1 = [0,\ 0,\ 2]^T$,$\alpha_1 = ||x_1|| = 2$,

$v_1 = x_1 - \alpha_1 e_1 = [-\ 2,\ 0,\ 2]^T$，$\beta_1 = \frac{1}{4}$

$$H_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \qquad H_1 A = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 4 & -2 \\ 0 & 3 & 1 \end{bmatrix}$$

取 $x_2 = [4, 3]^T$,

$\alpha_2 = ||x_2|| = 5$ $v_2 = x_2 - \alpha_2 e_2 = [-\ 1, 3]^T$，$\beta_2 = \frac{1}{5}$

$$\hat{H}_2 = \frac{1}{5}\begin{bmatrix} 4 & 3 \\ 3 & -4 \end{bmatrix}$$

$$H_2 = \begin{bmatrix} 1 & 0^T \\ 0 & \hat{H}_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{4}{5} & \frac{3}{5} \\ 0 & \frac{3}{5} & -\frac{4}{5} \end{bmatrix}$$

$$R = H_2(H_1A) = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 5 & -1 \\ 0 & 0 & -2 \end{bmatrix}$$

$$Q = H_1 H_2 = \frac{1}{5}\begin{bmatrix} 0 & 3 & -4 \\ 0 & 4 & 3 \\ 5 & 0 & 0 \end{bmatrix}$$

- Findings from the lab work

vitis library的host program 使用了xcl2的library，相比原本lab3的cl.h。cl.h的function比較底層，xcl2則整合了許多cl.h的function，簡化一些步驟，更方便使用者使用OpenCL的API。

Code explanation:

host：test_geqrf.cpp、matrixUtility.hpp、logger.hpp

kernel function ：kernel_geqrf.cpp、xf_solver_L2.hpp、geqrf.hpp

Host Program:

input data 是使用matGen產生(numRow = numCol = 16)

```
    // Generate general matrix numRow x numCol
    matGen<double>(numRow, numCol, seed, dataA_qrd);
void matGen(int m, int n, unsigned int seed, dataType* matrix) {
    srand(seed);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i * n + j] = (dataType)((rand() % (10000)) / (0.7 + n +
seed / 7));
        }
    }
}
```

執行kernel

```
// Setup kernel
    kernel_geqrf_0.setArg(0, input_buffer[0]);
    kernel_geqrf_0.setArg(1, output_buffer[0]);
    q.finish();
    std::cout << "INFO: Finish kernel setup" << std::endl;
    q.enqueueTask(kernel_geqrf_0, nullptr, nullptr);

    q.finish();
    std::cout << "INFO: Finish kernel execution" << std::endl;
```

透過kernel算出來的global mem(ob_out)傳回host mem的dataA_qrd(R矩陣)與tau_qrd(β)，再算出Q矩陣，並兩者相乘與原本的A矩陣比較。

```cpp
// Calculate A_out = Q*R and compare with original A matrix
    double* Q = new double[numRow * numRow];
    constructQ<double>(dataA_qrd, tau_qrd, numRow, numCol, Q);

    convertToRInline<double>(dataA_qrd, numRow, numCol);

    double* A = new double[numRow * numCol];
    matrixMult<double>(Q, numRow, numRow, dataA_qrd, numRow, numCol, A);

    bool equal = compareMatrices<double>(A, dataA, numRow, numCol, numCol);

    if (equal) {
        logger.info(xf::common::utils_sw::Logger::Message::TEST_PASS);
    } else {
        logger.error(xf::common::utils_sw::Logger::Message::TEST_FAIL);
    }
```

kernel function：

設定好矩陣後，進行運算。

```cpp
extern "C" void kernel_geqrf_0(double* dataA, double* tau) {
    // matrix initialization
    double dataA_2D[MA][NA];

    // Matrix transform from 1D to 2D
    int k = 0;
    for (int i = 0; i < MA; ++i) {
        for (int j = 0; j < NA; ++j) {
#pragma HLS pipeline
            dataA_2D[i][j] = dataA[k];
            k++;
        }
    }

    // Calling for QR
    xf::solver::geqrf<double, MA, NA, NCU>(MA, NA, dataA_2D, NA, tau);

    // Matrix transform from 2D to 1D
    k = 0;
    for (int i = 0; i < MA; ++i) {
        for (int j = 0; j < NA; ++j) {
#pragma HLS pipeline
            dataA[k] = dataA_2D[i][j];
            k++;
        }
    }
}
```

將data reshape以方便平行運算

```cpp
int geqrf(int m, int n, T A[NRMAX][NCMAX], int lda, T tau[NCMAX]) {
    static T data[NCU][NRMAX][(NCMAX + NCU - 1) / NCU];
    // read
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            data[j % NCU][i][j / NCU] = A[i][j];
        }
    }
    xf::solver::internal::qrf<T, NRMAX, NCMAX, NCU>(m, n, data, lda, tau);
    // write
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            A[i][j] = data[j % NCU][i][j / NCU];
        }
    }
    return 0;
}
```

將input data 分割成4份(NCU=4)做householder變換，計算出v向量和beta值，再將A矩陣更新。

```
template <typename DataType, int M, int N, int K>
void qrf(int m, int n, DataType matrix[K][M][(N + K - 1) / K], int lda,
DataType tau[N]) {
    DataType v[K][M];
#pragma HLS array_partition variable = v cyclic factor = K
loop_col:
    for (int i = 0; i < num; ++i) { // i: column index
        int cIdx = i % K;
        int cp = i / K;
        // create Householder vector
        const int length = m - i;
        DataType sum[16];
    loop_init:
        for (int j = 0; j < 16; ++j) {
#pragma HLS unroll
            sum[j] = 0.0;
        }
    loop_sum:
        for (int k = 1; k < length; ++k) {
#pragma HLS pipeline
#pragma HLS dependence variable = sum inter false
            sum[k % 16] += matrix[cIdx][k + i][cp] * matrix[cIdx][k + i][cp];
        }
        DataType s1[8];
        for (int idx = 0; idx < 8; ++idx) {
#pragma HLS pipeline
            int id = idx << 1;
            DataType temp = sum[id] + sum[id + 1];
            s1[idx] = temp;
        }
        DataType s2[4];
        for (int idx = 0; idx < 4; ++idx) {
#pragma HLS pipeline
            int id = idx << 1;
            s2[idx] = s1[id] + s1[id + 1];
        }
        DataType s3[2];
        for (int idx = 0; idx < 2; ++idx) {
```

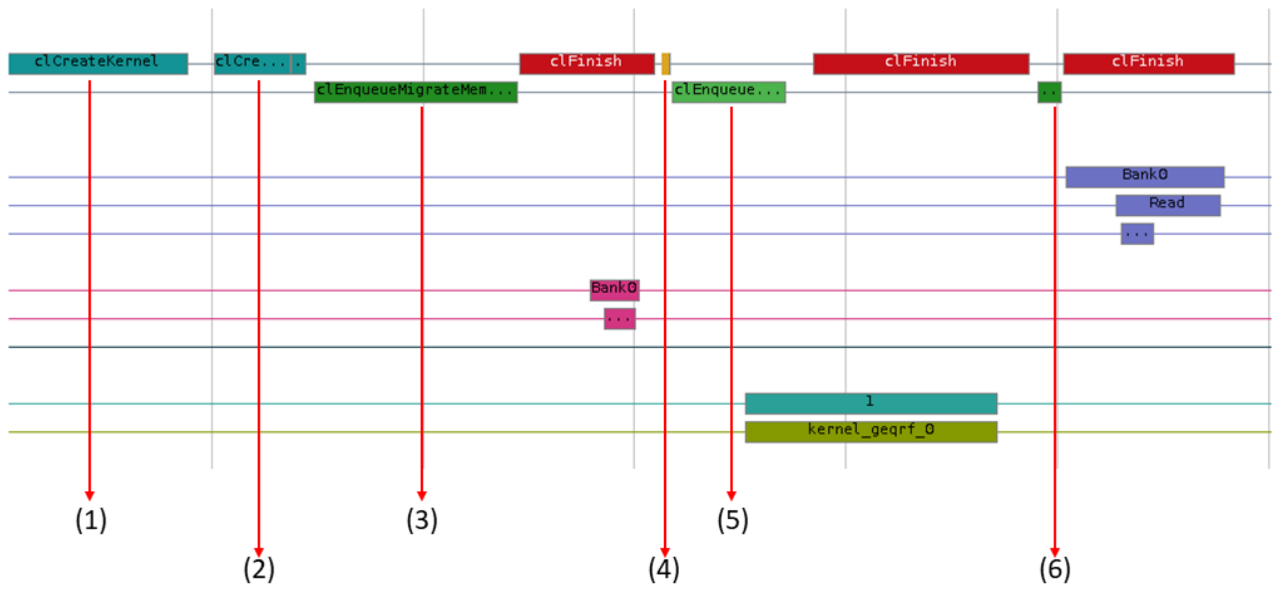
```
#pragma HLS pipeline
            int id = idx << 1;
            s3[idx] = s2[id] + s2[id + 1];
        }
        DataType accum1 = s3[0] + s3[1];

        DataType accum = matrix[cIdx][i][cp] * matrix[cIdx][i][cp] + accum1;

        DataType norm = xf::solver::internal::m::sqrt(accum);

        int sign = matrix[cIdx][i][cp] < 0 ? -1 : 1;
        matrix[cIdx][i][cp] -= sign * norm;
        DataType temp = matrix[cIdx][i][cp];
        if (xf::solver::internal::m::fabs(temp) > epsilon) {
        loop_vScale:
            for (int k = 0; k < length; ++k) {
#pragma HLS pipeline
#pragma HLS dependence variable = matrix inter false
                matrix[cIdx][k + i][cp] = matrix[cIdx][k + i][cp] / temp;
            }
        }
    loop_copyV:
        for (int k = 0; k < length; ++k) {
#pragma HLS pipeline
            for (int kk = 0; kk < K; ++kk) {
                v[kk][k] = matrix[cIdx][k + i][cp];
            }
        }
        DataType accumV = 1;
        if (xf::solver::internal::m::fabs(temp) > epsilon) {
            accumV += accum1 / (temp * temp);
        }
        DataType beta = 0.0;
        if (length > 1 && xf::solver::internal::m::fabs(accumV) > epsilon) {
            beta = 2.0 / accumV;
        }
        // store beta
        tau[i] = beta;
        update<DataType, M, N, K>(matrix, m, n, v, beta, i);
        matrix[cIdx][i][cp] = sign * norm;
    }
}
```

- OpenCL Timeline trace



(1) CreatKernel

(2) CreatBuffer

(3) enqueueMigrateMemObjects

(migrate from host to device)

(4) setKernelArgument

(5) enqueueTask(enqueue kernel)

(6) enqueueMigrateMemObjects

(migrate from device to host)

- Suggestion for improvement

如果矩陣大一點的話，也許可以使用stream的interface，原本在vitis library裡面是使用axi master的interface。因為Host program也只使用一個kernel,且只執行一次，我們想不太到其他optimization，memory transection也因為資料小的原因，read/write次數只有2次與3次Host read/write bandwidth可能也因為這樣非常的小，CU device utilization 在HW-emulation也到93%，更改NCU(number of compute unit)後也不會比較好。

- Encountered problem

因為Host program寫法不一樣，一開始我們再跑 run emulation時，一直讀不到xclbin。解決方法為在編輯Program Arguments時，在xclbin file前加入"-xclbin "就能讓Host讀到xclbin了。

- Github link

https://github.com/sssh311318/Lab_C_solver_QRF