ECE 641 – Lab for High Performance DSP

Term Project

GPU vs. CPU

Onur Yilmaz

31094453

Introduction

- The goal is to compare the GPU and CPU
- For which matrix size, CPU or GPU
- A parallel and iterative algorithm are needed.
- Jacobi's algorithm for Eigen Decomposition is suitable for comparing GPU and CPU
- Eigen Decomposition is generally used for graphic rendering, game programming etc.
- It has a high time complexity O(N^3)

Jacobi's Iterative Algorithm

$$A = J^T A J$$

```
void ClassicalJacobi(double pA[][SIZE], double pV[][SIZE]) {
    int index = 0;
    double eps = T * CalculateAbsolute(pA);

    while (CalculateOff(pA) > eps) {
        for (int p = 0; p < SIZE - 1; p++) {
            for (int q = p + 1; q < SIZE; q++) {
                double c = -1;
                double s = -1;
                double J[SIZE][SIZE];
                double JT[SIZE][SIZE];
                double JT_A[SIZE][SIZE];
                double JT_A_J[SIZE][SIZE];
                double V_J[SIZE][SIZE];

                SymmetricSchurDecomposition(pA, p, q, &c, &s);
                FormMatrixJ(J, p, q, c, s);
                CalculateTranspose(J, JT);
                MatrixMultiply(JT, pA, JT_A);
                MatrixMultiply(JT_A, J, JT_A_J);
                CopyMatrix(JT_A_J, pA);
                MatrixMultiply(pV, J, V_J);
                CopyMatrix(V_J, pV);
                index++;
            }
        }
    }
}
```

Jacobi's Iterative Algorithm – Calculate Absolute and Off

```
double CalculateOff(const double matrix[][SIZE]) {
    double offValue = 0.0;
    double total = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (i != j) {
                total += matrix[i][j] * matrix[i][j];
            }
        }
    }
    offValue = sqrt(total);
    return offValue;
}

double CalculateAbsolute(const double matrix[][SIZE]) {
    double offValue = 0.0;
    double total = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            total += matrix[i][j] * matrix[i][j];
        }
    }
    offValue = sqrt(total);
    return offValue;
}
```

Jacobi's Iterative Algorithm – Symmetric Shur Decomposition

```
void SymmetricSchurDecomposition(const double matrix[][SIZE],
                                 const int p, const int q, double *c,
                                 double *s) {
    if (matrix[p][q] == 0) {
        *c = 1;
        *s = 0;
    } else {
        twoMultiply = 2.0 * matrix[p][q];
        diff = matrix[q][q] - matrix[p][p];
        r = (double) ((double) diff / (double) twoMultiply);

        if (r >= 0.0) {
            r2 = r * r + 1;
            tSub = sqrt(r2);
            tSub = r + tSub;
            t = 1.0 / tSub;
        } else {
            r2 = r * r + 1;
            tSub = sqrt(r2);
            tSub = tSub - r;
            t = -1.0 / tSub;
        }
        t2 = t * t + 1;
        tSR = sqrt(t2);
        *c = 1 / tSR;
        *s = (*c) * t;
    }
}
```

Jacobi's Parallel Algorithm

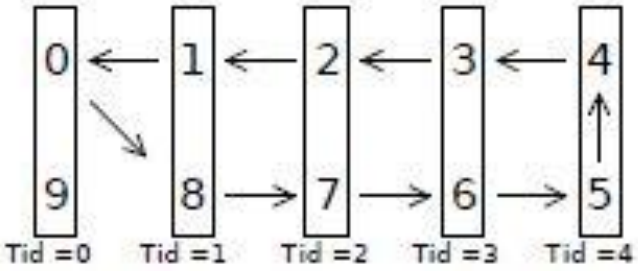


Fig. 1. Chess tournament ordering

- Idea is pairs can play in parallel because they do not disturb each other.
- Therefore, Symmetric Shur Decomposition and Matrix Multiplication can be performed in parallel

Jacobi's Parallel Algorithm

```
while (CalculateOff(pA) > eps) {
    double *pqs = ParallelSymShur(pA, pairs);
    for (int j=0; j<SIZE/2; j++){
        if (pairMatrix[0][j] < pairMatrix[1][j]) {
            p = pairMatrix[0][j];
            q = pairMatrix[1][j];
        } else {
            p = pairMatrix[1][j];
            q = pairMatrix[0][j];
        }

        FormMatrixJ(J, p, q, pqs[j * 2], pqs[j * 2 + 1]);
        CalculateTranspose(J, JT);
        ParallelMatrixMultiplication(JT, pA, JT_A);
        ParallelMatrixMultiplication(JT_A, J, JT_A_J);
        CopyMatrix(JT_A_J, pA);
        ParallelMatrixMultiplication(pV, J, V_J);
        CopyMatrix(V_J, pV);

    }
    free(pqs);

    NextPeriod(pairs, pairMatrix);
}
```

Jacobi's Parallel Algorithm – Parallel Symmetric Shur Decomposition

```
__global__ void SymShur(double *pA, int *pairs, double *results) {

    int id = threadIdx.x * 2;
    int p = 0;
    int q = 0;
    if (pairs[id] > pairs[id + 1]) {
        p = pairs[id + 1];
        q = pairs[id];
    } else {
        p = pairs[id];
        q = pairs[id + 1];
    }
    int dataIndexPP = p * SIZE + p;
    int dataIndexQQ = q * SIZE + q;
    int dataIndexPQ = p * SIZE + q;

    double r = 0.0;
    double twoMultiply = 0.0;
    double diff = 0.0;
    double t = 0.0;
    double r2 = 0.0;
    double t2 = 0.0;
    double tSub = 0.0;
    double tSR = 0.0;
    double c = 0.0;
    double s = 0.0;
```

Jacobi's Parallel Algorithm – Parallel Symmetric Shur Decomposition Continued

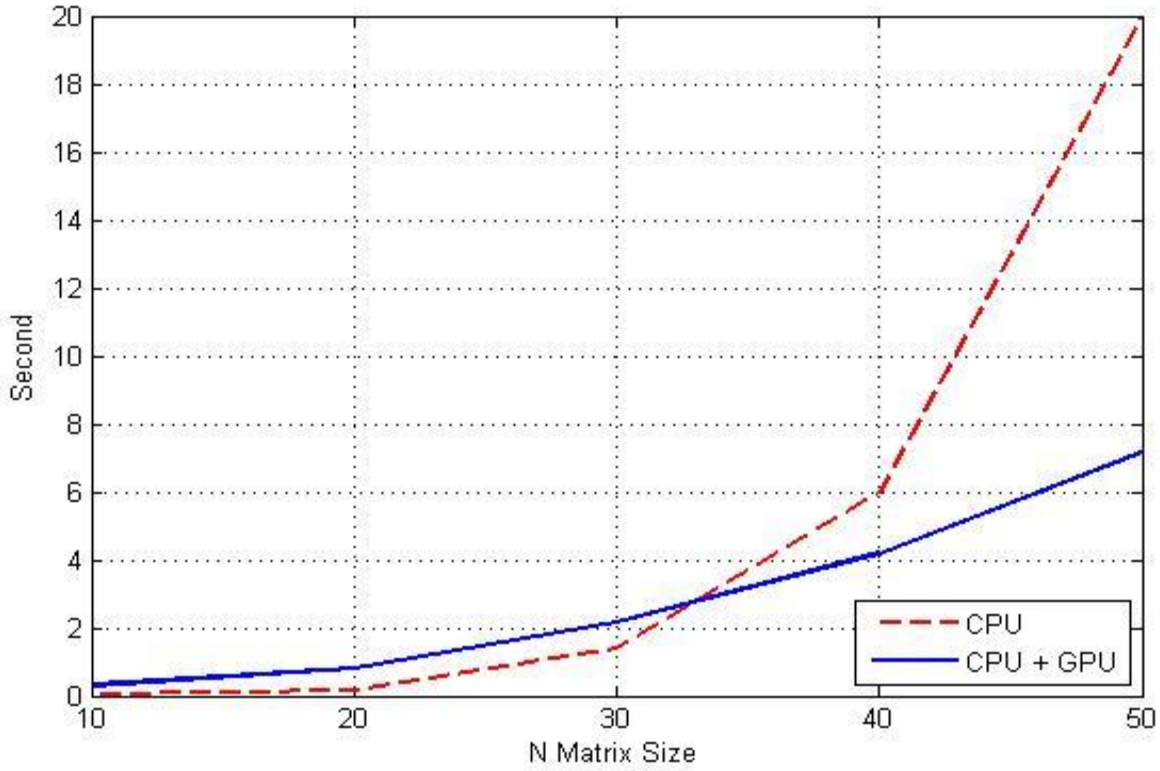
```
    if (pA[dataIndexPQ] == 0) {
        c = 1.0;
        s = 0.0;
    } else {
        twoMultiply = 2.0 * pA[dataIndexPQ];
        diff = pA[dataIndexQQ] - pA[dataIndexPP];
        r = (double) ((double) diff / (double) twoMultiply);

        if (r >= 0.0) {
            r2 = r * r + 1.0;
            tSub = sqrt(r2);
            tSub = r + tSub;
            t = 1.0 / tSub;
        } else {
            r2 = r * r + 1.0;
            tSub = sqrt(r2);
            tSub = tSub - r;
            t = -1.0 / tSub;
        }

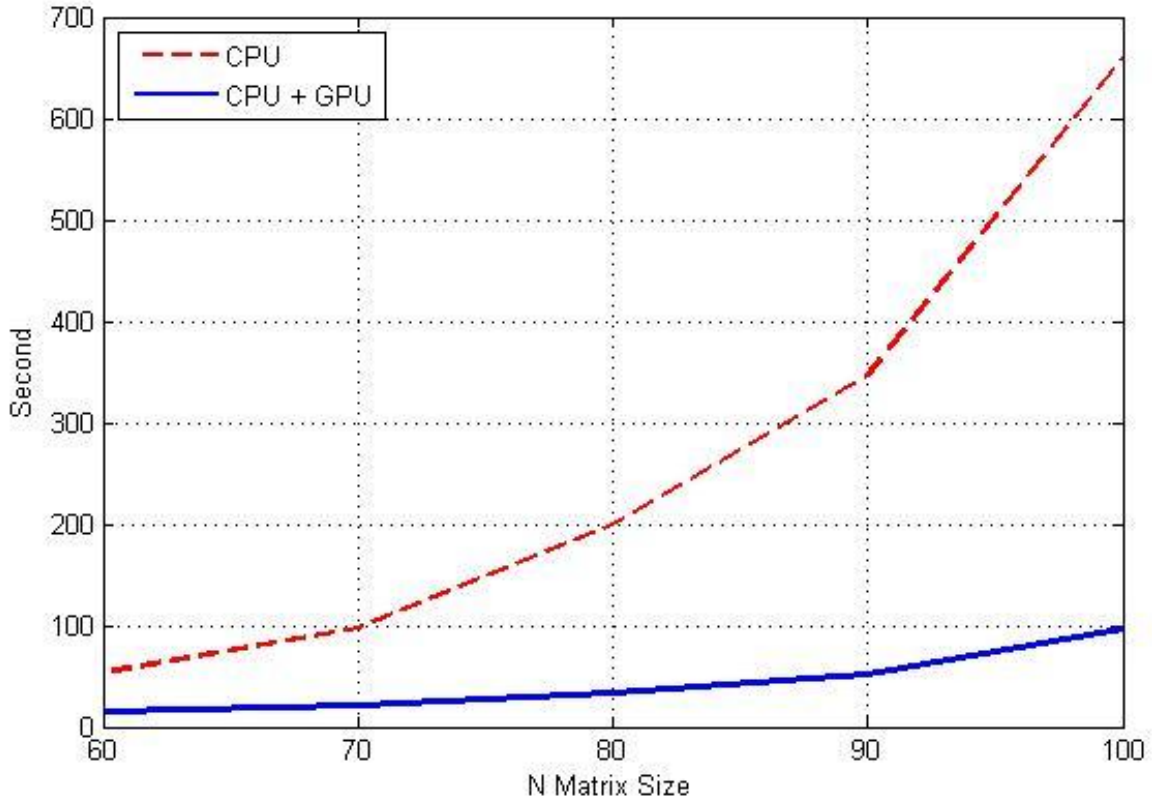
        t2 = t * t + 1.0;
        tSR = sqrt(t2);
        c = 1.0 / tSR;
        s = c * t;
    }

    results[id] = c;
    results[id + 1] = s;
}
```

Running Time Results



Running Time Results



Thank you.
Questions?