**DEPARTMENT OF MATHEMATICS AND COMPUTING**

**V-M.Tech. (M&C)**

**Monsoon Semester 2022-2023**

**GPU Computing Lab**

**MCC302**

**LAB-4**

**Matrix-Matrix Multiplication**

NAME: **ROMEO SARKAR**

ADMISSION NO.: **20JE0814**

DATE: **31-08-2022**

**Experiment 2.1:** Matrix-Matrix multiplication on GPU.

**Objectives:** Multiply two matrices.

**CUDA Sample Program:**

#include <stdio.h>

#include <cuda\_runtime.h>

#define N 3

\_\_global\_\_ void MatrixMulKernel (float \*MatA, float \*MatB, float \*MatC, int Width)

{

    int Row **=** blockIdx.y **\*** blockDim.y **+** threadIdx.y;

    int Col **=** blockIdx.x **\*** blockDim.x **+** threadIdx.x;

    if (Row **<** Width **&&** Col **<** Width)

    {

        // printf ("{%d,%d}", Row, Col);

        float Pvalue **=** 0;

        for (int k **=** 0; k **<** Width; k**++**)

        {

            // printf ("(%.0f,%.0f)", MatA[Row \* Width + k], MatB[k \* Width + Col]);

            Pvalue **+=** MatA[Row **\*** Width **+** k] **\*** MatB[k **\*** Width **+** Col];

        }

        MatC[Row **\*** Width **+** Col] **=** Pvalue;

        // printf ("=<%f>\n", Pvalue);

    }

}

void initialData (float \*ip, const int size)

{

    // int i;

    for (int i **=** 0; i **<** size; i**++**)

    {

        ip[i] **=** i;

    }

}

void displayMatrix (float \*A, int nx, int ny, int widthField)

{

    int idx;

    for (int i **=** 0; i **<** nx; i**++**)

    {

        for (int j **=** 0; j **<** ny; j**++**)

        {

            idx **=** i **\*** ny **+** j;

            printf (" %\*.0f ", widthField, A[idx]);

        }

        printf ("\n");

    }

}

int main ()

{

    int Width **=** N;

    int nx **=** Width;

    int ny **=** Width;

    int nxy **=** nx **\*** ny;

    int nBytes **=** nxy **\*** **sizeof** (float);

    printf ("Matrix size: nx %d ny %d\n", nx, ny);

    float **\***h\_A, **\***h\_B, **\***h\_C;

    h\_A **=** (float **\***) (malloc (nBytes));

    h\_B **=** (float **\***) malloc (nBytes);

    h\_C **=** (float **\***) malloc (nBytes);

    initialData (h\_A, nxy);

    initialData (h\_B, nxy);

    float **\***d\_MatA, **\***d\_MatB, **\***d\_MatC;

    cudaMalloc ((void **\*\***) **&**d\_MatA, nBytes);

    cudaMalloc ((void **\*\***) **&**d\_MatB, nBytes);

    cudaMalloc ((void **\*\***) **&**d\_MatC, nBytes);

    cudaMemcpy ((void **\***) d\_MatA, h\_A, nBytes, cudaMemcpyHostToDevice);

    cudaMemcpy ((void **\***) d\_MatB, h\_B, nBytes, cudaMemcpyHostToDevice);

    int bdimx **=** 16;

    int bdimy **=** 16;

    dim3 block (bdimx, bdimy, 1);

    dim3 grid ((nx **+** block.x **-** 1) **/** block.x, (ny **+** block.y **-** 1) **/** block.y, 1);

    MatrixMulKernel **<<<**grid, block**>>>** (d\_MatA, d\_MatB, d\_MatC, Width);

    cudaDeviceSynchronize ();

    cudaMemcpy (h\_C, d\_MatC, nBytes, cudaMemcpyDeviceToHost);

    printf ("Matrix A is=\n");

    displayMatrix (h\_A, nx, ny, 2);

    printf ("Matrix B is=\n");

    displayMatrix (h\_B, nx, ny, 2);

    printf ("The Product of Matrix A and Matrix B is=\n");

    displayMatrix (h\_C, nx, ny, 5);

    cudaFree (d\_MatA);

    cudaFree (d\_MatB);

    cudaFree (d\_MatC);

    free (h\_A);

    free (h\_B);

    free (h\_C);

    cudaDeviceReset ();

    return 0;

}

**Output:**

Matrix size: nx 3 ny 3

Matrix A is**=**

  0   1   2

  3   4   5

  6   7   8

Matrix B is**=**

  0   1   2

  3   4   5

  6   7   8

The Product of Matrix A **and** Matrix B is**=**

    15     18     21

    42     54     66

    69     90    111

**Lab Exercise 2.1:** Write a CUDA program to demonstrate the followings:

1. Allocate Device Memory.
2. Transfer Data (Matrices A, B and C) from host to device.
3. Find the product of three matrices A \* B \* C using 2D grid.
4. Transfer result from device to host.
5. Print the result in matrix format.

**CODE:**

#include <stdio.h>

#include <cuda\_runtime.h>

#define precisionField 0

\_\_global\_\_ void mul\_GPU (double \*m1, double \*m2, double \*p, int rows, int x, int cols);

void init (double \*p, int rows, int cols);

struct Matrix

{

    int rows, cols;

    double **\***device\_pointer, **\***host\_pointer;

    Matrix () : rows (0), cols (0), device\_pointer (NULL), host\_pointer (NULL)

    {

        return;

    }

    Matrix (int r, int c) : Matrix ()

    {

        rows **=** r;

        cols **=** c;

        alloc ();

        return;

    }

    Matrix (const Matrix **&**M)

    {

        rows **=** M.rows;

        cols **=** M.cols;

        cudaMalloc (**&**device\_pointer, rows **\*** cols **\*** **sizeof** (double));

        cudaMemcpy (device\_pointer, M.device\_pointer, rows **\*** cols **\*** **sizeof** (double), cudaMemcpyDeviceToDevice);

        host\_pointer **=** (double **\***) (malloc (rows **\*** cols **\*** **sizeof** (double)));

        memcpy (host\_pointer, M.host\_pointer, rows **\*** cols **\*** **sizeof** (double));

        return;

    }

    Matrix (Matrix **&&**M)

    {

        rows **=** M.rows;

        cols **=** M.cols;

        device\_pointer **=** M.device\_pointer;

        host\_pointer **=** M.host\_pointer;

        M.rows **=** M.cols **=** 0;

        M.device\_pointer **=** M.host\_pointer **=** NULL;

        return;

    }

    Matrix **operator =** (Matrix &M)

    {

        clear ();

        rows **=** M.rows;

        cols **=** M.cols;

        cudaMalloc (**&**device\_pointer, rows **\*** cols **\*** **sizeof** (double));

        cudaMemcpy (device\_pointer, M.device\_pointer, rows **\*** cols **\*** **sizeof** (double), cudaMemcpyDeviceToDevice);

        host\_pointer **=** (double **\***) (malloc (rows **\*** cols **\*** **sizeof** (double)));

        memcpy (host\_pointer, M.host\_pointer, rows **\*** cols **\*** **sizeof** (double));

        return **\***this;

    }

    Matrix **operator =** (Matrix &&M)

    {

        rows **=** M.rows;

        cols **=** M.cols;

        device\_pointer **=** M.device\_pointer;

        host\_pointer **=** M.host\_pointer;

        M.rows **=** M.cols **=** 0;

        M.device\_pointer **=** M.host\_pointer **=** NULL;

        return **\***this;

    }

**~**Matrix ()

    {

        if (NULL **!=** device\_pointer)

        {

            cudaFree (device\_pointer);

        }

        if (NULL **!=** host\_pointer)

        {

            free (host\_pointer);

        }

        rows **=** cols **=** 0;

        device\_pointer **=** host\_pointer **=** NULL;

        return;

    }

    void alloc ()

    {

        cudaMalloc (**&**device\_pointer, rows **\*** cols **\*** **sizeof** (double));

        host\_pointer **=** (double **\***) (malloc (rows **\*** cols **\*** **sizeof** (double)));

        return;

    }

    void clear ()

    {

        if (NULL **!=** device\_pointer)

        {

            cudaFree (device\_pointer);

        }

        if (NULL **!=** host\_pointer)

        {

            free (host\_pointer);

        }

        rows **=** cols **=** 0;

        device\_pointer **=** host\_pointer **=** NULL;

        return;

    }

    void display ()

    {

        int **\***max\_width\_arr **=** (int **\***) (malloc (cols **\*** **sizeof** (int)));

        char **\*\***mat\_of\_strs **=** (char **\*\***) malloc (rows **\*** cols **\*** **sizeof** (char **\***));

        char **\***str;

        int width;

        for (size\_t i **=** 0; i **<** cols; i**++**)

        {

            max\_width\_arr[i] **=** 1;

            for (size\_t j **=** 0; j **<** rows; j**++**)

            {

                str **=** (char **\***) malloc (128 **\*** **sizeof** (char));

                width **=** snprintf (str, 128, "%.\*lf", precisionField, host\_pointer[j **\*** cols **+** i]);

                str **=** (char **\***) realloc (str, ((size\_t) (width **+** 1)) **\*** **sizeof** (char));

                mat\_of\_strs[j **\*** cols **+** i] **=** str;

                if (max\_width\_arr[i] **<** width)

                    max\_width\_arr[i] **=** width;

            }

        }

        for (size\_t i **=** 0; i **<** rows; i**++**)

        {

            printf ("\xb3");

            for (size\_t j **=** 0; j **<** cols; j**++**)

            {

                width **=** strlen (mat\_of\_strs[i **\*** cols **+** j]);

                for (int x **=** 0; x **<** max\_width\_arr[j] **-** width; x**++**)

                    printf (" ");

                printf ("%s", mat\_of\_strs[i **\*** cols **+** j]);

                if (j **!=** (cols **-** 1))

                    printf (" ");

            }

            printf ("\xb3");

            // newline:

            printf ("\n");

        }

        for (size\_t i **=** 0; i **<** rows; i**++**)

            for (size\_t j **=** 0; j **<** cols; j**++**)

                free (mat\_of\_strs[i **\*** cols **+** j]);

        free (mat\_of\_strs);

        free (max\_width\_arr);

        return;

    }

    void init ()

    {

        ::init (host\_pointer, rows, cols);

        H2D ();

        return;

    }

    void H2D () // Transfer Data from host to device

    {

        cudaMemcpy (device\_pointer, host\_pointer, cols **\*** rows **\*** **sizeof** (double), cudaMemcpyHostToDevice);

        return;

    }

    void D2H () // Transfer Data from device to host

    {

        cudaMemcpy (host\_pointer, device\_pointer, cols **\*** rows **\*** **sizeof** (double), cudaMemcpyDeviceToHost);

        return;

    }

    Matrix **operator \*** (const Matrix &M)

    {

        if (cols **!=** M.rows)

        {

            printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);

            return Matrix ();

        }

        Matrix p (rows, M.cols);

        dim3 block (1, 1, 1);

        dim3 grid (rows, M.cols, 1);

        mul\_GPU **<<<**grid, block**>>>** (device\_pointer, M.device\_pointer, p.device\_pointer, rows, cols, M.cols);

        cudaDeviceSynchronize ();

        p.D2H ();

        return p;

    }

};

void init (double \*p, int rows, int cols)

{

    for (int i **=** 0; i **<** rows **\*** cols; i**++**)

    {

        p[i] **=** rand () **%** 21 **-** 10;

    }

    return;

}

\_\_global\_\_ void mul\_GPU (double \*m1, double \*m2, double \*p, int rows, int x, int cols)

{

    int Row **=** blockIdx.x **\*** blockDim.x **+** threadIdx.x;

    int Col **=** blockIdx.y **\*** blockDim.y **+** threadIdx.y;

    if (Row **<** rows **&&** Col **<** cols)

    {

        double a **=** 0;

        for (int k **=** 0; k **<** x; k**++**)

        {

            a **+=** m1[Row **\*** x **+** k] **\*** m2[k **\*** cols **+** Col];

        }

        p[Row **\*** cols **+** Col] **=** a;

    }

    return;

}

int main ()

{

    Matrix A (4, 4), B (4, 4), C (4, 4);

    A.init (), B.init (), C.init ();

    printf ("Matrix A:\n");

    A.display ();

    printf ("Matrix B:\n");

    B.display ();

    printf ("Matrix C:\n");

    C.display ();

    Matrix D **=** A **\*** B **\*** C;

    printf ("Matrix D (A \* B \* C):\n");

    D.display ();

    cudaDeviceReset ();

    return 0;

}

**Output:**

Matrix A:

│10 **-**2  3   9│

│ 7  6  2 **-**10│

│ 9 10  4  **-**5│

│ 3 **-**4 **-**3  **-**2│

Matrix B:

│ 3  4  8  8│

│**-**1 **-**1  7 **-**4│

│ 9  3  2 **-**5│

│10 **-**2 **-**2  2│

Matrix C:

│**-**2 **-**1 **-**10 **-**6│

│**-**5 **-**3  **-**6 **-**7│

│ 7  8   1 **-**1│

│ 2 **-**8 **-**10 **-**6│

Matrix D (A \* B \* C):

│ 97 **-**544 **-**2544 **-**1725│

│752  883   484   **-**68│

│878 1117  **-**178  **-**526│

│ 73 **-**455  **-**242  **-**173│

**Lab Exercise 2.2:** Write a CUDA program to demonstrate:

1. Allocate Device Memory.
2. Transfer Data (Matrices A and B) from host to device.
3. Find the transpose (TA and TB) of matrices A and B in parallel on GPU.
4. Find the product of A and B and TA and TB.
5. Transfer results from device to host.
6. Print the result matrices and their differences.

**CODE:**

#include <stdio.h>

#include <cuda\_runtime.h>

#define precisionField 0

struct Matrix;

\_\_global\_\_ void mul\_GPU (double \*m1, double \*m2, double \*p, int rows, int x, int cols);

\_\_global\_\_ void trp\_GPU (double \*m1, double \*m2, int rows, int cols);

\_\_global\_\_ void sub\_GPU (double \*m1, double \*m2, double \*a, int rows, int cols);

void init (double \*p, int rows, int cols);

struct Matrix

{

    int rows, cols;

    double **\***device\_pointer, **\***host\_pointer;

    int flag **=** 0;

    Matrix () : rows (0), cols (0), device\_pointer (NULL), host\_pointer (NULL)

    {

        return;

    }

    Matrix (int r, int c) : Matrix ()

    {

        rows **=** r;

        cols **=** c;

        alloc ();

        return;

    }

    Matrix (const Matrix **&**M)

    {

        rows **=** M.rows;

        cols **=** M.cols;

        cudaMalloc (**&**device\_pointer, rows **\*** cols **\*** **sizeof** (double));

        cudaMemcpy (device\_pointer, M.device\_pointer, rows **\*** cols **\*** **sizeof** (double), cudaMemcpyDeviceToDevice);

        host\_pointer **=** (double **\***) (malloc (rows **\*** cols **\*** **sizeof** (double)));

        memcpy (host\_pointer, M.host\_pointer, rows **\*** cols **\*** **sizeof** (double));

        return;

    }

    Matrix (Matrix **&&**M)

    {

        rows **=** M.rows;

        cols **=** M.cols;

        device\_pointer **=** M.device\_pointer;

        host\_pointer **=** M.host\_pointer;

        M.rows **=** M.cols **=** 0;

        M.device\_pointer **=** M.host\_pointer **=** NULL;

        return;

    }

    Matrix **operator =** (Matrix &M)

    {

        clear ();

        rows **=** M.rows;

        cols **=** M.cols;

        cudaMalloc (**&**device\_pointer, rows **\*** cols **\*** **sizeof** (double));

        cudaMemcpy (device\_pointer, M.device\_pointer, rows **\*** cols **\*** **sizeof** (double), cudaMemcpyDeviceToDevice);

        host\_pointer **=** (double **\***) (malloc (rows **\*** cols **\*** **sizeof** (double)));

        memcpy (host\_pointer, M.host\_pointer, rows **\*** cols **\*** **sizeof** (double));

        return **\***this;

    }

    Matrix **operator =** (Matrix &&M)

    {

        rows **=** M.rows;

        cols **=** M.cols;

        device\_pointer **=** M.device\_pointer;

        host\_pointer **=** M.host\_pointer;

        M.rows **=** M.cols **=** 0;

        M.device\_pointer **=** M.host\_pointer **=** NULL;

        return **\***this;

    }

**~**Matrix ()

    {

        if (NULL **!=** device\_pointer)

        {

            cudaFree (device\_pointer);

        }

        if (NULL **!=** host\_pointer)

        {

            free (host\_pointer);

        }

        rows **=** cols **=** 0;

        device\_pointer **=** host\_pointer **=** NULL;

        return;

    }

    void alloc ()

    {

        cudaMalloc (**&**device\_pointer, rows **\*** cols **\*** **sizeof** (double));

        host\_pointer **=** (double **\***) (malloc (rows **\*** cols **\*** **sizeof** (double)));

        return;

    }

    void clear ()

    {

        if (NULL **!=** device\_pointer)

        {

            cudaFree (device\_pointer);

        }

        if (NULL **!=** host\_pointer)

        {

            free (host\_pointer);

        }

        rows **=** cols **=** 0;

        device\_pointer **=** host\_pointer **=** NULL;

        return;

    }

    void display ()

    {

        int **\***max\_width\_arr **=** (int **\***) (malloc (cols **\*** **sizeof** (int)));

        char **\*\***mat\_of\_strs **=** (char **\*\***) malloc (rows **\*** cols **\*** **sizeof** (char **\***));

        char **\***str;

        int width;

        for (size\_t i **=** 0; i **<** cols; i**++**)

        {

            max\_width\_arr[i] **=** 1;

            for (size\_t j **=** 0; j **<** rows; j**++**)

            {

                str **=** (char **\***) malloc (128 **\*** **sizeof** (char));

                width **=** snprintf (str, 128, "%.\*lf", precisionField, host\_pointer[j **\*** cols **+** i]);

                str **=** (char **\***) realloc (str, ((size\_t) (width **+** 1)) **\*** **sizeof** (char));

                mat\_of\_strs[j **\*** cols **+** i] **=** str;

                if (max\_width\_arr[i] **<** width)

                    max\_width\_arr[i] **=** width;

            }

        }

        for (size\_t i **=** 0; i **<** rows; i**++**)

        {

            printf ("\xb3");

            for (size\_t j **=** 0; j **<** cols; j**++**)

            {

                width **=** strlen (mat\_of\_strs[i **\*** cols **+** j]);

                for (int x **=** 0; x **<** max\_width\_arr[j] **-** width; x**++**)

                    printf (" ");

                printf ("%s", mat\_of\_strs[i **\*** cols **+** j]);

                if (j **!=** (cols **-** 1))

                    printf (" ");

            }

            printf ("\xb3");

            // newline:

            printf ("\n");

        }

        for (size\_t i **=** 0; i **<** rows; i**++**)

            for (size\_t j **=** 0; j **<** cols; j**++**)

                free (mat\_of\_strs[i **\*** cols **+** j]);

        free (mat\_of\_strs);

        free (max\_width\_arr);

        return;

    }

    void init ()

    {

        ::init (host\_pointer, rows, cols);

        H2D ();

        return;

    }

    void H2D ()

    {

        cudaMemcpy (device\_pointer, host\_pointer, cols **\*** rows **\*** **sizeof** (double), cudaMemcpyHostToDevice);

        return;

    }

    void D2H ()

    {

        cudaMemcpy (host\_pointer, device\_pointer, cols **\*** rows **\*** **sizeof** (double), cudaMemcpyDeviceToHost);

        return;

    }

    Matrix **operator -** (const Matrix &M)

    {

        if (rows **!=** M.rows **&&** cols **!=** M.cols)

        {

            printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);

            return Matrix ();

        }

        Matrix p (rows, M.cols);

        dim3 block (1, 1, 1);

        dim3 grid (rows, M.cols, 1);

        sub\_GPU **<<<** block, grid**>>>** (device\_pointer, M.device\_pointer, p.device\_pointer, rows, cols);

        cudaDeviceSynchronize ();

        p.D2H ();

        // p.display ();

        return p;

    }

    Matrix **operator \*** (const Matrix &M)

    {

        if (cols **!=** M.rows)

        {

            printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);

            return Matrix ();

        }

        Matrix p (rows, M.cols);

        dim3 block (1, 1, 1);

        dim3 grid (rows, M.cols, 1);

        mul\_GPU **<<<** block, grid**>>>** (device\_pointer, M.device\_pointer, p.device\_pointer, rows, cols, M.cols);

        cudaDeviceSynchronize ();

        p.D2H ();

        // p.display ();

        return p;

    }

    Matrix **operator ~** ()

    {

        Matrix t (cols, rows);

        dim3 block (1, 1, 1);

        dim3 grid (rows, cols, 1);

        trp\_GPU **<<<**grid, block**>>>** (device\_pointer, t.device\_pointer, rows, cols);

        cudaDeviceSynchronize ();

        t.D2H ();

        return t;

    }

};

void init (double \*p, int rows, int cols)

{

    for (int i **=** 0; i **<** rows **\*** cols; i**++**)

    {

        p[i] **=** rand () **%** 21 **-** 10;

    }

    return;

}

\_\_global\_\_ void sub\_GPU (double \*m1, double \*m2, double \*a, int rows, int cols)

{

    int Row **=** blockIdx.x **\*** blockDim.x **+** threadIdx.x;

    int Col **=** blockIdx.y **\*** blockDim.y **+** threadIdx.y;

    if (Row **<** rows **&&** Col **<** cols)

    {

        a[Row **\*** cols **+** Col] **=** m1[Row **\*** cols **+** Col] **-** m2[Row **\*** cols **+** Col];

    }

    return;

}

\_\_global\_\_ void mul\_GPU (double \*m1, double \*m2, double \*p, int rows, int x, int cols)

{

    int Row **=** blockIdx.x **\*** blockDim.x **+** threadIdx.x;

    int Col **=** blockIdx.y **\*** blockDim.y **+** threadIdx.y;

    if (Row **<** rows **&&** Col **<** cols)

    {

        double a **=** 0;

        for (int k **=** 0; k **<** x; k**++**)

        {

            a **+=** m1[Row **\*** x **+** k] **\*** m2[k **\*** cols **+** Col];

        }

        p[Row **\*** cols **+** Col] **=** a;

    }

    return;

}

\_\_global\_\_ void trp\_GPU (double \*m1, double \*m2, int rows, int cols)

{

    int Row **=** blockIdx.x **\*** blockDim.x **+** threadIdx.x;

    int Col **=** blockIdx.y **\*** blockDim.y **+** threadIdx.y;

    if (Row **<** rows **&&** Col **<** cols)

    {

        m2[Col **\*** rows **+** Row] **=** m1[Row **\*** cols **+** Col];

    }

    return;

}

int main ()

{

    srand (time (NULL));

    Matrix A (4, 4), B (4, 4);

    A.init (), B.init ();

    Matrix TA **=** **~**A, TB **=** **~**B;

    printf ("Matrix A:\n");

    A.display ();

    printf ("Matrix B:\n");

    B.display ();

    printf ("Matrix TA:\n");

    TA.display ();

    printf ("Matrix TB:\n");

    TB.display ();

    Matrix AB **=** A **\*** B;

    Matrix TATB **=** TA **\*** TB;

    printf ("Matrix AB:\n");

    AB.display ();

    printf ("Matrix TATB:\n");

    TATB.display ();

    Matrix D **=** AB **-** TATB;

    printf ("Matrix AB - TATB:\n");

    D.display ();

    cudaDeviceReset ();

    return 0;

}

**Outputs:**

Matrix A:

│ 3  5  1 **-**2│

│ 6  3 **-**6 10│

│ 6 10  8  0│

│**-**2  0  9  4│

Matrix B:

│**-**6  6 **-**9 **-**7│

│ 4 **-**7 **-**3 **-**5│

│**-**8  1  1  9│

│ 2 **-**6  3  6│

Matrix TA:

│ 3  6  6 **-**2│

│ 5  3 10  0│

│ 1 **-**6  8  9│

│**-**2 10  0  4│

Matrix TB:

│**-**6  4 **-**8  2│

│ 6 **-**7  1 **-**6│

│**-**9 **-**3  1  3│

│**-**7 **-**5  9  6│

Matrix AB:

│**-**10  **-**4 **-**47 **-**49│

│ 44 **-**51 **-**39 **-**51│

│**-**60 **-**26 **-**76 **-**20│

│**-**52 **-**27  39 119│

Matrix TATB:

│ **-**22 **-**38 **-**30 **-**24│

│**-**102 **-**31 **-**27  22│

│**-**177 **-**23  75 116│

│  44 **-**98  62 **-**40│

Matrix AB **-** TATB:

│ 12  34  **-**17  **-**25│

│146 **-**20  **-**12  **-**73│

│117  **-**3 **-**151 **-**136│

│**-**96  71  **-**23  159│