```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

```
!pip install nvcc4jupyter
```

```
Collecting nvcc4jupyter
  Downloading nvcc4jupyter-1.2.1-py3-none-any.whl.metadata (5.1 kB)
Downloading nvcc4jupyter-1.2.1-py3-none-any.whl (10 kB)
Installing collected packages: nvcc4jupyter
Successfully installed nvcc4jupyter-1.2.1
```

```
%load_ext nvcc4jupyter
```

```
Detected platform "Colab". Running its setup...
Source files will be saved in "/tmp/tmpxqr2k3fu".
```

OpenMP

1. Write an OpenMP program to calculate the value of PI using critical section

```
%%writefile criticalpi.c

#include<stdio.h>
#include<omp.h>
#define n 1000000
double step;
int main() {
    int i;
    double x, pi, sum=0.0, start_time, end_time;
    start_time=omp_get_wtime();
    step=1.0/(double)n;
    #pragma omp critical
    {
        for(i=0;i<n;i++) {
            x=(i+0.5)*step;
            sum=sum+4.0/(1.0+x*x);
        }
    }
    end_time = omp_get_wtime();
    pi = step*sum;
    printf("Calculated value of PI is %f\n", pi);
    printf("Time taken is %f\n", end_time-start_time);
    return 0;
}
```

```
Writing criticalpi.c
```

```
!gcc -fopenmp criticalpi.c
```

```
!./a.out
```

```
Calculated value of PI is 3.141593
Time taken is 0.007671
```

```
%%writefile criticalpi2.c

#include <omp.h>
#include <stdio.h>

int main() {
    long long num_steps = 100000000;  // Number of iterations for precision
    double pi = 0.0;
    double step = 1.0 / (double)num_steps;

    // Start time measurement
    double start_time = omp_get_wtime();

    // Start the parallel region
    #pragma omp parallel
```

```
    {
        double sum = 0.0;  // Private sum for each thread

        // Calculate the local sum for each thread
        #pragma omp for
        for (long long i = 0; i < num_steps; i++) {
            double x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }

        // Use a critical section to accumulate each thread's sum into pi
        #pragma omp critical
        {
            pi += sum * step;
        }
    }  // End of parallel region

    // End time measurement
    double end_time = omp_get_wtime();

    // Calculate and print the time taken and the result
    printf("Calculated value of Pi: %.15f\n", pi);
    printf("Time taken: %f seconds\n", end_time - start_time);

    return 0;
}
```

```
Writing criticalpi2.c
```

```
!gcc -fopenmp criticalpi2.c
```

```
!./a.out
```

```
Calculated value of Pi: 3.141592653589909
Time taken: 0.300474 seconds
```

2. Write an OpenMP program to print parallel programming environment information

```
%%writefile envi_info.c

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int nthreads, tid, procs, maxt, inpar, dynamic, nested;
    #pragma omp parallel private(nthreads, tid)
    {
    tid = omp_get_thread_num();
        if (tid == 0)
        {
            printf("thread %d getting envi info \n",tid);
            procs = omp_get_num_procs();
            nthreads = omp_get_num_threads();
            maxt = omp_get_max_threads();
            inpar = omp_in_parallel();
            dynamic = omp_get_dynamic();
            nested = omp_get_nested();

            printf("Number of processors = %d\n", procs);
            printf("Number of threads = %d\n", nthreads);
            printf("Max threads = %d\n", maxt);
            printf("In Parellel= %d ",inpar);
            printf("Dynamic = %d\n", dynamic);
            printf("Nested = %d\n", nested);
        }
    }
}
```

```
Writing envi_info.c
```

```
!gcc -fopenmp envi_info.c
```

```
!./a.out
```

```
thread 0 getting envi info
Number of processors = 2
Number of threads = 2
Max threads = 2
In Parellel= 1 Dynamic = 0
Nested = 0
```

3. Write an OpenMP program to add 2 arrays parallely using dynamic clause

```
%%writefile dynamic.c

#include <stdio.h>
#include <omp.h>

#define N 10  // Size of the arrays

int main() {
    int A[N], B[N], C[N];  // Declare arrays A, B, and C
    int i;
    // Initialize arrays A and B with values
    for (i = 0; i < N; i++) {
        A[i] = i + 1;
        B[i] = (i + 1) * 2;
    }

    // Perform parallel addition of arrays A and B into C
    #pragma omp parallel for shared(A, B, C) schedule(dynamic) private(i)
    for (i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
        printf("Thread %d is processing element %d\n", omp_get_thread_num(), i);
    }

    // Verify the result by printing the first few elements
    printf("First 10 elements of the result array C:\n");
    for (i = 0; i < 10; i++) {
        printf("C[%d] = %d\n", i, C[i]);
    }

    return 0;
}
```

```
Overwriting dynamic.c
```

```
!gcc -fopenmp dynamic.c
```

```
!./a.out
```

```
Thread 0 is processing element 0
Thread 0 is processing element 2
Thread 0 is processing element 3
Thread 0 is processing element 4
Thread 0 is processing element 5
Thread 0 is processing element 6
Thread 0 is processing element 7
Thread 0 is processing element 8
Thread 0 is processing element 9
Thread 1 is processing element 1
First 10 elements of the result array C:
C[0] = 3
C[1] = 6
C[2] = 9
C[3] = 12
C[4] = 15
C[5] = 18
C[6] = 21
C[7] = 24
C[8] = 27
C[9] = 30
```

4. Write an OpenMP program to add and multiply 2 arrays with 2 different threads

```
%%writefile addmul.c

#include <stdio.h>
#include <omp.h>

#define SIZE 5
```

```
int main() {
    int a[SIZE] = {1,2,3,4,5};
    int b[SIZE] = {5,4,3,2,1};
    int c[SIZE], d[SIZE];
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            int tid = omp_get_thread_num();
            printf("Addition section executed by thread = %d\n", tid);
            for(int i=0; i<SIZE; i++) {
                c[i] = a[i] + b[i];
                printf("c[%d] = %d\n", i, c[i]);
            }
        }
        #pragma omp section
        {
            int tid = omp_get_thread_num();
            printf("Multiplication section executed by thread = %d\n", tid);
            for(int i=0; i<SIZE; i++) {
                d[i] = a[i] * b[i];
                printf("d[%d] = %d\n", i, d[i]);
            }
        }
    }
    return 0;
}
```

```
Writing addmul.c
```

```
!gcc -fopenmp addmul.c
```

```
!./a.out
```

```
Addition section executed by thread = 1
c[0] = 6
c[1] = 6
c[2] = 6
c[3] = 6
c[4] = 6
Multiplication section executed by thread = 0
d[0] = 5
d[1] = 8
d[2] = 9
d[3] = 8
d[4] = 5
```

5. Write an OpenMP program to perform matrix multiplication

```
%%writefile openmp_mul.c

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

#define NRA 62
#define NCA 15
#define NCB 7

int main(int argc, char* argv[]){
    int tid, nthreads, i, j, k, chunk;
    double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];
    chunk = 10;
    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid, i, j, k)
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Starting matrix multiplication example with %d threads\n", nthreads);
        printf("Initializing matrices...\n");
    }
    #pragma omp for schedule(static, chunk)
    for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
            a[i][j]= i+j;
    #pragma omp for schedule(static, chunk)
    for (i=0; i<NCA; i++)
        for (j=0; j<NCB; j++)
            b[i][j]= i*j;
    #pragma omp for schedule(static, chunk)
```

```
    for (i=0; i<NRA; i++)
        for (j=0; j<NCB; j++)
            c[i][j]= 0;
    printf("Thread %d starting matrix multiplication\n", tid);
    #pragma omp for schedule(static, chunk)
    for(i=0; i<NRA; i++){
        printf("Thread %d did row = %d\n", tid, i);
        for(j=0; j<NCB; j++){
            for(k=0; k<NCA; k++){
                c[i][j] += a[i][k]*b[k][j];
            }
        }
    }
    printf("Thread %d done. Result matrix:\n", tid);
    for(i=0; i<NRA; i++){
        for(j=0; j<NCB; j++){
            printf("%6.2f", c[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

```
Overwriting openmp_mul.c
```

```
!gcc -fopenmp openmp_mul.c
```

```
!./a.out
```

```
Starting matrix multiplication example with 1 threads
Initializing matrices...
Thread 0 starting matrix multiplication
Thread 0 did row = 0
Thread 0 did row = 1
Thread 0 did row = 2
Thread 0 did row = 3
Thread 0 did row = 4
Thread 0 did row = 5
Thread 0 did row = 6
Thread 0 did row = 7
Thread 0 did row = 8
Thread 0 did row = 9
Thread 0 did row = 10
Thread 0 did row = 11
Thread 0 did row = 12
Thread 0 did row = 13
Thread 0 did row = 14
Thread 0 did row = 15
Thread 0 did row = 16
Thread 0 did row = 17
Thread 0 did row = 18
Thread 0 did row = 19
Thread 0 did row = 20
Thread 0 did row = 21
Thread 0 did row = 22
Thread 0 did row = 23
Thread 0 did row = 24
Thread 0 did row = 25
Thread 0 did row = 26
Thread 0 did row = 27
Thread 0 did row = 28
Thread 0 did row = 29
Thread 0 did row = 30
Thread 0 did row = 31
Thread 0 did row = 32
Thread 0 did row = 33
Thread 0 did row = 34
Thread 0 did row = 35
Thread 0 did row = 36
Thread 0 did row = 37
Thread 0 did row = 38
Thread 0 did row = 39
Thread 0 did row = 40
Thread 0 did row = 41
Thread 0 did row = 42
Thread 0 did row = 43
Thread 0 did row = 44
Thread 0 did row = 45
Thread 0 did row = 46
Thread 0 did row = 47
Thread 0 did row = 48
Thread 0 did row = 49
Thread 0 did row = 50
Thread 0 did row = 51
Thread 0 did row = 52
Thread 0 did row = 53
```

```
Thread 0 did row = 54
```

6. Write an OpenMP program to demonstrate first private clause

```
%%writefile firstpvt.c

#include <omp.h>
#include <stdio.h>

int main() {
    int x = 10;  // Shared variable
    int result = 0;  // Variable to accumulate results

    printf("Initial value of x: %d\n", x);

    // Parallel region
    #pragma omp parallel firstprivate(x)
    {
        int tid = omp_get_thread_num();  // Thread ID
        x += tid;  // Each thread modifies its private copy of x
        printf("Thread %d: x = %d\n", tid, x);

        // Use reduction to safely sum results
        #pragma omp critical
        {
            result += x;  // Accumulate private x values into shared result
        }
    }

    printf("Final accumulated result: %d\n", result);
    return 0;
}
```

```
Writing firstpvt.c
```

```
!gcc -fopenmp firstpvt.c
```

```
!./a.out
```

```
Initial value of x: 10
Thread 0: x = 10
Thread 1: x = 11
Final accumulated result: 21
```

7. Write an OpenMP program to add all the numbers in a vector by demonstrating the use of the reduction clause.

```
%%writefile reduction.c

#include <stdio.h>
#include <omp.h>

int main() {
    int i;
    int vector[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < 10; i++) {
        sum += vector[i];
    }

    printf("Sum of vector elements: %d\n", sum);

    return 0;
}
```

```
Writing reduction.c
```

```
!gcc -fopenmp reduction.c
```

```
!./a.out
```

```
Sum of vector elements: 55
```

CUDA

Write a CUDA program to add 2 numbers

```
%%cuda

#include <stdio.h>

// CUDA kernel to add two numbers
__global__ void addNumbers(int *a, int *b, int *c) {
    *c = *a + *b; // Add the two numbers
}

int main() {
    int a = 5, b = 7; // Numbers to add
    int c;            // Result of addition

    int *d_a, *d_b, *d_c; // Device pointers

    // Allocate memory on the device (GPU)
    cudaMalloc((void **)&d_a, sizeof(int));
    cudaMalloc((void **)&d_b, sizeof(int));
    cudaMalloc((void **)&d_c, sizeof(int));

    // Check if memory allocation was successful
    if (d_a == NULL || d_b == NULL || d_c == NULL) {
        printf("Failed to allocate device memory.\n");
        return -1;
    }

    // Copy input data from host to device
    cudaMemcpy(d_a, &a, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, sizeof(int), cudaMemcpyHostToDevice);

    // Launch the kernel (1 block, 1 thread)
    addNumbers<<<1, 1>>>(d_a, d_b, d_c);

    // Synchronize device to ensure kernel execution completes
    cudaDeviceSynchronize();

    // Copy the result from device to host
    cudaMemcpy(&c, d_c, sizeof(int), cudaMemcpyDeviceToHost);

    // Print the result
    printf("The sum of %d and %d is %d\n", a, b, c);

    // Free device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

```
The sum of 5 and 7 is 12
```

Write a CUDA program to perform vector addition

```
%%cuda

#include <stdio.h>
#include <cuda_runtime.h>
#define SIZE 64

__global__ void addVector(int *a, int *b, int *c) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < SIZE) {
        c[tid] = a[tid] + b[tid];
    }
}

void randomNumberAssigner(int *array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = rand() % 100;
```

```
    }
}

int main() {
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    printf("In main function\n");
    a = (int*) malloc(SIZE * sizeof(int));
    b = (int*) malloc(SIZE * sizeof(int));
    c = (int*) malloc(SIZE * sizeof(int));
    randomNumberAssigner(a, SIZE);
    randomNumberAssigner(b, SIZE);
    cudaMalloc((void**) &d_a, SIZE * sizeof(int));
    cudaMalloc((void**) &d_b, SIZE * sizeof(int));
    cudaMalloc((void**) &d_c, SIZE * sizeof(int));
    cudaMemcpy(d_a, a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, SIZE * sizeof(int), cudaMemcpyHostToDevice);
    addVector<<<2,32>>>(d_a, d_b, d_c);
    cudaMemcpy(c, d_c, SIZE * sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < SIZE; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(a);
    free(b);
    free(c);
    return 0;
}
```

Show hidden output

Write a program to perform matrix addition

```
%%cuda

#include <stdio.h>
#include <cuda_runtime.h>
#define ROW 10
#define COL 10

__global__ void addMatrix(int *a, int *b, int *c) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < ROW && col < COL) {
        c[row * COL + col] = a[row * COL + col] + b[row * COL + col];
    }
}
void randomNumberAssigner(int *array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = rand() % 10;
    }
}

int main() {
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = ROW * COL;
    printf("In main function\n");
    a = (int*) malloc(size * sizeof(int));
    b = (int*) malloc(size * sizeof(int));
    c = (int*) malloc(size * sizeof(int));
    randomNumberAssigner(a, size);
    randomNumberAssigner(b, size);
    cudaMalloc((void**) &d_a, size * sizeof(int));
    cudaMalloc((void**) &d_b, size * sizeof(int));
    cudaMalloc((void**) &d_c, size * sizeof(int));
    cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
    dim3 threadsPerBlock(COL, ROW);
    dim3 numBlocks(1,1);
    // int tbp = 64, nb = ((ROW*COL)+tbp-1)/tbp;
    addMatrix<<<threadsPerBlock, numBlocks>>>(d_a, d_b, d_c);
    cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
        printf("Matrix A:\n");
        for (int i = 0; i < ROW; i++) {
          for(int j = 0; j < COL; j++)
            printf("%d\t", a[i*COL+j]);
          printf("\n");
```

```
        }
        printf("Matrix B:\n");
        for (int i = 0; i < ROW; i++) {
          for(int j = 0; j < COL; j++)
            printf("%d\t", b[i*COL+j]);
          printf("\n");
        }
        printf("Matrix C:\n");
        for (int i = 0; i < ROW; i++) {
            for (int j = 0; j < COL; j++)
                printf("%d\t", c[i*COL+j]);
            printf("\n");
        }
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(a);
    free(b);
    free(c);
    return 0;
}
```

```
In main function
Matrix A:
3	6	7	5	3	5	6	2	9	1
2	7	0	9	3	6	0	6	2	6
1	8	7	9	2	0	2	3	7	5
9	2	2	8	9	7	3	6	1	2
9	3	1	9	4	7	8	4	5	0
3	6	1	0	6	3	2	0	6	1
5	5	4	7	6	5	6	9	3	7
4	5	2	5	4	7	4	4	3	0
7	8	6	8	8	4	3	1	4	9
2	0	6	8	9	2	6	6	4	9
Matrix B:
5	0	4	8	7	1	7	2	7	2
2	6	1	0	6	1	5	9	4	9
0	9	1	7	7	1	1	5	9	7
7	6	7	3	6	5	6	3	9	4
8	1	2	9	3	9	0	8	8	5
0	9	6	3	8	5	6	1	1	5
9	8	4	8	1	0	3	0	4	4
4	4	7	6	3	1	7	5	9	6
2	1	7	8	5	7	4	1	8	5
9	7	5	3	8	8	3	1	8	9
Matrix C:
8	6	11	13	10	6	13	4	16	3
4	13	1	9	9	7	5	15	6	15
1	17	8	16	9	1	3	8	16	12
16	8	9	11	15	12	9	9	10	6
17	4	3	18	7	16	8	12	13	5
3	15	7	3	14	8	8	1	7	6
14	13	8	15	7	5	9	9	7	11
8	9	9	11	7	8	11	9	12	6
9	9	13	16	13	11	7	2	12	14
11	7	11	11	17	10	9	7	12	18
```

Write a CUDA program to perform matrix multiplication.

```
%%cuda
#include <stdio.h>
#include <cuda_runtime.h>
#define ROW_A 10
#define COL_A 5
#define ROW_B COL_A
#define COL_B 8
__global__ void matrixMul(int *a, int *b, int *c) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < ROW_A && col < COL_B) {
        int sum = 0;
        for (int k = 0; k < COL_A; k++) {
            sum += a[row * COL_A + k] * b[k * COL_B + col];
        }
        c[row * COL_B + col] = sum;
    }
}
void randomNumberAssigner(int *array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = rand() % 10;
    }
}
```

```
int main() {
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size_a = ROW_A * COL_A;
    int size_b = ROW_B * COL_B;
    int size_c = ROW_A * COL_B;
    printf("In main function\n");
    a = (int*) malloc(size_a * sizeof(int));
    b = (int*) malloc(size_b * sizeof(int));
    c = (int*) malloc(size_c * sizeof(int));
    randomNumberAssigner(a, size_a);
    randomNumberAssigner(b, size_b);
    cudaMalloc((void**) &d_a, size_a * sizeof(int));
    cudaMalloc((void**) &d_b, size_b * sizeof(int));
    cudaMalloc((void**) &d_c, size_c * sizeof(int));
    cudaMemcpy(d_a, a, size_a * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size_b * sizeof(int), cudaMemcpyHostToDevice);
    //dim3 threadsPerBlock(COL, ROW);
    //dim3 numBlocks(1,1);
    int tbp = 32;
    dim3 threadsPerBlock(tbp, tbp);
    dim3 numBlocks((COL_B+threadsPerBlock.x-1)/threadsPerBlock.x, (ROW_A+threadsPerBlock.y-1)/threadsPerBlock.y);
    matrixMul<<<numBlocks, threadsPerBlock>>>(d_a, d_b, d_c);
    cudaMemcpy(c, d_c, size_c * sizeof(int), cudaMemcpyDeviceToHost);
        printf("Matrix A:\n");
        for (int i = 0; i < ROW_A; i++) {
          for(int j = 0; j < COL_A; j++)
            printf("%d\t", a[i*COL_A+j]);
          printf("\n");
        }
        printf("Matrix B:\n");
        for (int i = 0; i < ROW_B; i++) {
          for(int j = 0; j < COL_B; j++)
            printf("%d\t", b[i*COL_B+j]);
          printf("\n");
        }
        printf("Matrix C:\n");
        for (int i = 0; i < ROW_A; i++) {
            for (int j = 0; j < COL_B; j++)
                printf("%d\t", c[i*COL_B+j]);
            printf("\n");
        }
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(a);
    free(b);
    free(c);
    return 0;
}
```

```
In main function
Matrix A:
3	6	7	5	3
5	6	2	9	1
2	7	0	9	3
6	0	6	2	6
1	8	7	9	2
0	2	3	7	5
9	2	2	8	9
7	3	6	1	2
9	3	1	9	4
7	8	4	5	0
Matrix B:
3	6	1	0	6	3	2	0
6	1	5	5	4	7	6	5
6	9	3	7	4	5	2	5
4	7	4	4	3	0	7	8
6	8	8	4	3	1	4	9
Matrix C:
125	146	98	111	94	89	103	132
105	125	85	84	92	68	117	121
102	106	97	83	76	58	121	134
98	152	80	74	84	54	62	100
141	156	114	133	99	96	135	165
88	118	87	79	56	34	87	126
137	202	129	92	121	60	126	165
91	122	60	69	87	74	59	71
111	161	95	74	109	57	117	128
113	121	79	88	105	97	105	100
```

Write a CUDA program to perform following operations:

- Take 2 matrices A, B
- Find the transpose TA, TB
- Perform C = A*B, TC = TA*TB
- Verify whether C and TC are equal or not.

```
%%cuda
#include <stdio.h>
#include <cuda_runtime.h>

#define ROW_A 2
#define COL_A 3
#define ROW_B COL_A
#define COL_B 2


__global__ void transpose(int *input, int *output, int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < rows && col < cols) {
        output[col * rows + row] = input[row * cols + col];
    }
}


__global__ void matrixMul(int *a, int *b, int *c, int row_a, int col_a, int col_b) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < row_a && col < col_b) {
        int sum = 0;
        for (int k = 0; k < col_a; k++) {
            sum += a[row * col_a + k] * b[k * col_b + col];
        }
        c[row * col_b + col] = sum;
    }
}


int main() {
    int a[ROW_A * COL_A] = {1, 2, 3, 4, 5, 6};
    int b[ROW_B * COL_B] = {7, 8, 9, 10, 11, 12};
    int c[ROW_A * COL_B], tc[ROW_A * COL_B];
    int ta[COL_A * ROW_A], tb[COL_B * ROW_B];


    int *d_a, *d_b, *d_c, *d_ta, *d_tb, *d_tc;

    cudaMalloc(&d_a, ROW_A * COL_A * sizeof(int));
    cudaMalloc(&d_b, ROW_B * COL_B * sizeof(int));
    cudaMalloc(&d_c, ROW_A * COL_B * sizeof(int));
    cudaMalloc(&d_ta, COL_A * ROW_A * sizeof(int));
    cudaMalloc(&d_tb, COL_B * ROW_B * sizeof(int));
    cudaMalloc(&d_tc, ROW_A * COL_B * sizeof(int));

    cudaMemcpy(d_a, a, ROW_A * COL_A * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, ROW_B * COL_B * sizeof(int), cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocksA((COL_A + threadsPerBlock.x - 1) / threadsPerBlock.x, (ROW_A + threadsPerBlock.y-1) / threadsPerBlock.y);
    dim3 numBlocksB((COL_B + threadsPerBlock.x - 1) / threadsPerBlock.x, (ROW_B + threadsPerBlock.y-1) / threadsPerBlock.y);
    dim3 numBlocksC((COL_B + threadsPerBlock.x - 1) / threadsPerBlock.x, (ROW_A + threadsPerBlock.y - 1) / threadsPerBlock.y);

    transpose<<<numBlocksA, threadsPerBlock>>>(d_a, d_ta, ROW_A, COL_A);
    transpose<<<numBlocksB, threadsPerBlock>>>(d_b, d_tb, ROW_B, COL_B);


    matrixMul<<<numBlocksC, threadsPerBlock>>>(d_a, d_b, d_c, ROW_A, COL_A, COL_B);
    matrixMul<<<numBlocksC, threadsPerBlock>>>(d_ta, d_tb, d_tc, COL_A, ROW_A, ROW_B);

    cudaMemcpy(c, d_c, ROW_A * COL_B * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(tc, d_tc, ROW_A * COL_B * sizeof(int), cudaMemcpyDeviceToHost);

    //Verification
    printf("Matrix A:\n");
    for (int i = 0; i < ROW_A; i++){
      for(int j = 0; j < COL_A; j++) {
        printf("%d\t", a[i*COL_A+j]);
      }
      printf("\n");
```

```
    }

    printf("Matrix B:\n");
    for (int i = 0; i < ROW_B; i++){
      for(int j = 0; j < COL_B; j++) {
        printf("%d\t", b[i*COL_B+j]);
      }
      printf("\n");
    }

    printf("Matrix C:\n");
    for (int i = 0; i < ROW_A; i++){
      for(int j = 0; j < COL_B; j++) {
        printf("%d\t", c[i*COL_B+j]);
      }
      printf("\n");
    }

    printf("\nMatrix TC:\n");
    for(int i = 0; i < ROW_A; i++) {
      for (int j = 0; j < COL_B; j++){
        printf("%d\t", tc[i * COL_B + j]);
      }
      printf("\n");
    }

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFree(d_ta);
    cudaFree(d_tb);
    cudaFree(d_tc);
    return 0;
}
```

```
Matrix A:
1       2       3
4       5       6
Matrix B:
7       8
9       10
11      12
Matrix C:
58      64
139     154

Matrix TC:
39      49
59      54
```

Write a CUDA program to perform dot product on two vectors.

```
%%cuda

#include <stdio.h>
#include <cuda_runtime.h>

#define SIZE 10

__global__ void dotProduct(int *a, int *b, int *result) {
    __shared__ int partial_sums[256]; // Shared memory for partial sums
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    int sum = 0;
    if(i < SIZE) {
        sum = a[i] * b[i];
    }

    partial_sums[tid] = sum;
    __syncthreads(); // Synchronize threads within the block

    // Perform reduction in shared memory
    for(int s = blockDim.x / 2; s > 0; s >>= 1) {
        if(tid < s) {
            partial_sums[tid] += partial_sums[tid + s];
        }
        __syncthreads();
    }
```

```
    if (tid == 0) {
        atomicAdd(result, partial_sums[0]);
    }
}


int main() {
    int *a, *b, result = 0;
    int *d_a, *d_b, *d_result;

    a = (int*)malloc(SIZE * sizeof(int));
    b = (int*)malloc(SIZE * sizeof(int));

    for (int i = 0; i < SIZE; i++) {
        a[i] = i;
        b[i] = i * 2;
    }

    cudaMalloc(&d_a, SIZE * sizeof(int));
    cudaMalloc(&d_b, SIZE * sizeof(int));
    cudaMalloc(&d_result, sizeof(int));

    cudaMemcpy(d_a, a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, SIZE * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemset(d_result, 0, sizeof(int)); // Initialize result to 0 on the device


    int threadsPerBlock = 256;
    int blocksPerGrid = (SIZE + threadsPerBlock -1) / threadsPerBlock;
    dotProduct<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_result);


    cudaMemcpy(&result, d_result, sizeof(int), cudaMemcpyDeviceToHost);

    printf("\nVector A:\n");
    for(int i = 0; i < SIZE; i++) {
        printf("%d\t", a[i]);
    }
    printf("\n");

    printf("\nVector B:\n");
    for(int i = 0; i < SIZE; i++) {
        printf("%d\t", b[i]);
    }
    printf("\n");

    printf("\nDot Product: %d\n", result);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_result);
    free(a);
    free(b);

    return 0;
}
```

```
Vector A:
0       1       2       3       4       5       6       7       8       9

Vector B:
0       2       4       6       8       10      12      14      16      18

Dot Product: 570
```

Start coding or generate with AI.