```
%%writefile vector.cu
#include<stdio.h>
#define N 512
//#include<iostream>
__global__ void addVectors(int* A, int* B, int* C)
{
    C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];
}


int main(int argc,char **argv)
{
    //int n = 100;
    int* A, * B, * C;
    int size = N * sizeof(int);

    // Allocate memory on the host
    cudaMallocHost(&A, size);
    cudaMallocHost(&B, size);
    cudaMallocHost(&C, size);

    // Initialize the vectors
    for (int i = 0; i < N; i++)
    {
        A[i] = i;
        B[i] = i * 2;
    }
    // Allocate memory on the device
    int* dev_A, * dev_B, * dev_C;
    cudaMalloc(&dev_A, size);
    cudaMalloc(&dev_B, size);
    cudaMalloc(&dev_C, size);

    // Copy data from host to device
    cudaMemcpy(dev_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, size, cudaMemcpyHostToDevice);

    // Launch the kernel

    //int blockSize = 16;
    //int numBlocks = (n + blockSize - 1) / blockSize;
  addVectors<<<1,N>>>(dev_A, dev_B, dev_C);
    // Copy data from device to host
    cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);

    // Print the results
    for (int i = 0; i < N; i++)
    {
        printf("\n %d",C[i]);
    }
    //cout << endl;

    // Free memory
    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
    cudaFreeHost(A);
    cudaFreeHost(B);
    cudaFreeHost(C);

    return 0;
}
```

```
Overwriting vector.cu
```

```
!nvcc vector.cu -o vector
```

```
!./vector
```

```
192
195
198
201
204
207
210
213
216
219
222
225
228
231
234
237
240
243
246
249
252
255
258
261
264
267
270
273
276
279
282
285
288
291
294
297
300
303
306
309
312
315
318
321
324
327
330
333
336
339
342
345
348
351
```

```
!nvidia-smi
```

```
Mon Apr  1 05:34:15 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   40C    P8              9W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

```
%%writefile hello.cu
```

```
#include<stdio.h>
__global__ void hello(void)
{
    printf("GPU: Hello!\n");
}
int main(int argc,char **argv)
{
    printf("CPU: Hello!\n");
    hello<<<1,10>>>();
    cudaDeviceReset();
    return 0;
}
```

```
 Writing hello.cu
```

```
!nvcc hello.cu -o hello
```

```
!./hello
```

```
CPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
```

```
!which nvcc
```

```
/usr/local/cuda/bin/nvcc
```

```
%%writefile matrix.cu
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <cuda.h>

#define BLOCK_SIZE 16

__global__ void gpu_matrix_mult(int *a,int *b, int *c, int n)

{

int row = blockIdx.y * blockDim.y + threadIdx.y;

int col = blockIdx.x * blockDim.x + threadIdx.x;

int sum = 0;

if( col < n && row < n)

{

for(int i = 0; i < n; i++)

{

sum += a[row * n + i] * b[i * n + col];

}

c[row * n + col] = sum;

}
```

```
}

int main(int argc, char const *argv[])

{

int n;

for(n = 64; n <=8192 ; n *= 2) {

// allocate memory in host RAM, h_cc is used to store CPU result

int *h_a, *h_b, *h_c, *h_cc;

cudaMallocHost((void **) &h_a, sizeof(int)*n*n);

cudaMallocHost((void **) &h_b, sizeof(int)*n*n);

cudaMallocHost((void **) &h_c, sizeof(int)*n*n);

// initialize matrix A

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

h_a[i * n + j] = 2;

}

}

// initialize matrix B

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

h_b[i * n + j] = 3;

}

}

float naive_gpu_elapsed_time_ms;

// some events to count the execution time

//clock_t st, end;

cudaEvent_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// Allocate memory space on the device

int *d_a, *d_b, *d_c;

cudaMalloc((void **) &d_a, sizeof(int)*n*n);

cudaMalloc((void **) &d_b, sizeof(int)*n*n);

cudaMalloc((void **) &d_c, sizeof(int)*n*n);

// copy matrix A and B from host to device memory

cudaMemcpy(d_a, h_a, sizeof(int)*n*n, cudaMemcpyHostToDevice);

cudaMemcpy(d_b, h_b, sizeof(int)*n*n, cudaMemcpyHostToDevice);

unsigned int grid_rows = (n + 15) / BLOCK_SIZE;

unsigned int grid_cols = (n + 15) / BLOCK_SIZE;
```

```
dim3 dimGrid(grid_cols, grid_rows);

dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

cudaEventRecord(start, 0);

gpu_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, n);

cudaThreadSynchronize();

// time counting terminate

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

// Transfer results from device to host

cudaMemcpy(h_cc, d_c, sizeof(int)*n*n, cudaMemcpyDeviceToHost);

// compute time elapsed on GPU computing

cudaEventElapsedTime(&naive_gpu_elapsed_time_ms, start, stop);

printf("Time elapsed on naive GPU matrix multiplication of %dx%d . %dx%d : %f ms.\n\n", n, n, n, n, naive_gpu_elapsed_time_ms)

// free memory

cudaFree(d_a);

cudaFree(d_b);

cudaFree(d_c);



}

return 0;

`
```

```
Writing matrix.cu
```

```
!nvcc matrix.cu -o matrix
```

```
matrix.cu(131): warning #549-D: variable "h_cc" is used before its value is set
  cudaMemcpy(h_cc, d_c, sizeof(int)*n*n, cudaMemcpyDeviceToHost);
             ^

Remark: The warnings can be suppressed with "-diag-suppress <warning-number>"

matrix.cu(131): warning #549-D: variable "h_cc" is used before its value is set
  cudaMemcpy(h_cc, d_c, sizeof(int)*n*n, cudaMemcpyDeviceToHost);
             ^

Remark: The warnings can be suppressed with "-diag-suppress <warning-number>"

matrix.cu: In function ‘int main(int, const char**)’:
matrix.cu:121:22: warning: ‘cudaError_t cudaThreadSynchronize()’ is deprecated [-Wdeprecated-declarations]
  121 | cudaThreadSynchronize();
      | ~~~~~~~~~~~~~~~~~~~~~^~
/usr/local/cuda/bin/../targets/x86_64-linux/include/cuda_runtime_api.h:1069:46: note: declared here
 1069 | extern __CUDA_DEPRECATED __host__ cudaError_t CUDARTAPI cudaThreadSynchronize(void);
      |                                              ^~~~~~~~~~~~~~~~~~~~
```

```
!./matrix
```

```
Time elapsed on naive GPU matrix multiplication of 64x64 . 64x64 : 0.249632 ms.

Time elapsed on naive GPU matrix multiplication of 128x128 . 128x128 : 0.043040 ms.

Time elapsed on naive GPU matrix multiplication of 256x256 . 256x256 : 0.174144 ms.

Time elapsed on naive GPU matrix multiplication of 512x512 . 512x512 : 1.159776 ms.

Time elapsed on naive GPU matrix multiplication of 1024x1024 . 1024x1024 : 9.139872 ms.
```

```
Time elapsed on naive GPU matrix multiplication of 2048x2048 . 2048x2048 : 74.979614 ms.

Time elapsed on naive GPU matrix multiplication of 4096x4096 . 4096x4096 : 403.352570 ms.

Time elapsed on naive GPU matrix multiplication of 8192x8192 . 8192x8192 : 2299.902832 ms.
```

```
%%writefile v.cu
#include <assert.h>
#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <time.h>


#define N (1024*1024)
#define M (1000000)

void random_ints(int* a, int N)
{
   int i;
   for (i = 0; i < M; ++i)
    a[i] = rand() %5000;
}


__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
    }


    int main(void) {
    int *a, *b, *c;     // host copies of a, b, c
    int *d_a, *d_b, *d_c;   // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU with N blocks
        add<<<N,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
    }
```

```
Overwriting v.cu
```

```
!nvcc v.cu -o v
```

```
v.cu(12): error: expected a ")"
  void random_ints(int* a, int (1024*1024))
                               ^

1 error detected in the compilation of "v.cu".
```

```
%%writefile mat.cu
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define BLOCK_SIZE 16

/*
*********************************************************************
function name: gpu_matrix_mult

description: dot product of two matrix (not only square)

parameters:
            &a GPU device pointer to a m X n matrix (A)
            &b GPU device pointer to a n X k matrix (B)
            &c GPU device output purpose pointer to a m X k matrix (C)
            to store the result

Note:
    grid and block should be configured as:
        dim3 dimGrid((k + BLOCK_SIZE - 1) / BLOCK_SIZE, (m + BLOCK_SIZE - 1) / BLOCK_SIZE);
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

    further sppedup can be obtained by using shared memory to decrease global memory access times
return: none
*********************************************************************
*/
__global__ void gpu_matrix_mult(int *a,int *b, int *c, int m, int n, int k)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if( col < k && row < m)
    {
        for(int i = 0; i < n; i++)
        {
            sum += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = sum;
    }
}

/*
*********************************************************************
function name: gpu_square_matrix_mult

description: dot product of two matrix (not only square) in GPU

parameters:
            &a GPU device pointer to a n X n matrix (A)
            &b GPU device pointer to a n X n matrix (B)
            &c GPU device output purpose pointer to a n X n matrix (C)
            to store the result
Note:
    grid and block should be configured as:

        dim3 dim_grid((n - 1) / BLOCK_SIZE + 1, (n - 1) / BLOCK_SIZE + 1, 1);
        dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE, 1);

return: none
*********************************************************************
*/
__global__ void gpu_square_matrix_mult(int *d_a, int *d_b, int *d_result, int n)
{
    __shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int tmp = 0;
    int idx;

    for (int sub = 0; sub < gridDim.x; ++sub)
    {
        idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
        if(idx >= n*n)
```

```
        {
            // n may not divisible by BLOCK_SIZE
            tile_a[threadIdx.y][threadIdx.x] = 0;
        }
        else
        {
            tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
        }

        idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
        if(idx >= n*n)
        {
            tile_b[threadIdx.y][threadIdx.x] = 0;
        }
        else
        {
            tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
        }
        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
        }
        __syncthreads();
    }
    if(row < n && col < n)
    {
        d_result[row * n + col] = tmp;
    }
}

/*
*********************************************************************
function name: gpu_matrix_transpose

description: matrix transpose

parameters:
            &mat_in GPU device pointer to a rows X cols matrix
            &mat_out GPU device output purpose pointer to a cols X rows matrix
            to store the result
Note:
    grid and block should be configured as:
        dim3 dim_grid((n - 1) / BLOCK_SIZE + 1, (n - 1) / BLOCK_SIZE + 1, 1);
        dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE, 1);

return: none
*********************************************************************
*/
__global__ void gpu_matrix_transpose(int* mat_in, int* mat_out, unsigned int rows, unsigned int cols)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < cols && idy < rows)
    {
        unsigned int pos = idy * cols + idx;
        unsigned int trans_pos = idx * rows + idy;
        mat_out[trans_pos] = mat_in[pos];
    }
}
/*
*********************************************************************
function name: cpu_matrix_mult

description: dot product of two matrix (not only square) in CPU,
             for validating GPU results

parameters:
            &a CPU host pointer to a m X n matrix (A)
            &b CPU host pointer to a n X k matrix (B)
            &c CPU host output purpose pointer to a m X k matrix (C)
            to store the result
return: none
*********************************************************************
*/
```

```
void cpu_matrix_mult(int *h_a, int *h_b, int *h_result, int m, int n, int k) {
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int tmp = 0.0;
            for (int h = 0; h < n; ++h)
            {
                tmp += h_a[i * n + h] * h_b[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}

/*
**********************************************************************
function name: main

description: test and compare

parameters:
            none

return: none
**********************************************************************
*/
int main(int argc, char const *argv[])
{
    int m, n, k;
    /* Fixed seed for illustration */
    srand(3333);
    printf("please type in m n and k\n");
    scanf("%d %d %d", &m, &n, &k);

    // allocate memory in host RAM, h_cc is used to store CPU result
    int *h_a, *h_b, *h_c, *h_cc;
    cudaMallocHost((void **) &h_a, sizeof(int)*m*n);
    cudaMallocHost((void **) &h_b, sizeof(int)*n*k);
    cudaMallocHost((void **) &h_c, sizeof(int)*m*k);
    cudaMallocHost((void **) &h_cc, sizeof(int)*m*k);

    // random initialize matrix A
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            h_a[i * n + j] = rand() % 1024;
        }
    }

    // random initialize matrix B
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < k; ++j) {
            h_b[i * k + j] = rand() % 1024;
        }
    }

    float gpu_elapsed_time_ms, cpu_elapsed_time_ms;

    // some events to count the execution time
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // start to count execution time of GPU version
    cudaEventRecord(start, 0);
    // Allocate memory space on the device
    int *d_a, *d_b, *d_c;
    cudaMalloc((void **) &d_a, sizeof(int)*m*n);
    cudaMalloc((void **) &d_b, sizeof(int)*n*k);
    cudaMalloc((void **) &d_c, sizeof(int)*m*k);

    // copy matrix A and B from host to device memory
    cudaMemcpy(d_a, h_a, sizeof(int)*m*n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, sizeof(int)*n*k, cudaMemcpyHostToDevice);

    unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
    unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
```

```
    dim3 dimGrid(grid_cols, grid_rows);
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

    // Launch kernel
    if(m == n && n == k)
    {
        gpu_square_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, n);
    }
    else
    {
        gpu_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, m, n, k);
    }
    // Transefr results from device to host
    cudaMemcpy(h_c, d_c, sizeof(int)*m*k, cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    // time counting terminate
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    // compute time elapse on GPU computing
    cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on GPU: %f ms.\n\n", m, n, n, k, gpu_elapsed_time_ms)

    // start the CPU version
    cudaEventRecord(start, 0);

    cpu_matrix_mult(h_a, h_b, h_cc, m, n, k);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&cpu_elapsed_time_ms, start, stop);
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on CPU: %f ms.\n\n", m, n, n, k, cpu_elapsed_time_ms)

    // validate results computed by GPU
    int all_ok = 1;
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            printf("[%d][%d]:%d == [%d][%d]:%d, ", i, j, h_cc[i*k + j], i, j, h_c[i*k + j]);
            if(h_cc[i*k + j] != h_c[i*k + j])
            {
                all_ok = 0;
            }
        }
        //printf("\n");
    }

    // roughly compute speedup
    if(all_ok)
    {
        printf("all results are correct!!!, speedup = %f\n", cpu_elapsed_time_ms / gpu_elapsed_time_ms);
    }
    else
    {
        printf("incorrect results\n");
    }

    // free memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFreeHost(h_a);
    cudaFreeHost(h_b);
    cudaFreeHost(h_c);
```

```
Overwriting mat.cu
```

```
!nvcc mat.cu -o mat
```

```
mat.cu: In function ‘int main(int, const char**)’:
mat.cu:245:22: warning: ‘cudaError_t cudaThreadSynchronize()’ is deprecated [-Wdeprecated-declarations]
  245 |     cudaThreadSynchronize();
      |     ~~~~~~~~~~~~~~~~~^~
/usr/local/cuda/bin/../targets/x86_64-linux/include/cuda_runtime_api.h:1069:46: note: declared here
 1069 | extern __CUDA_DEPRECATED __host__ cudaError_t CUDARTAPI cudaThreadSynchronize(void);
      |                                              ^~~~~~~~~~~~~~~~~~~~
```

```
!./mat
```

```
     please type in m n and k
     1024 2048 2048
     Time elapsed on matrix multiplication of 1024x2048 . 2048x2048 on GPU: 41.400417 ms.

     Time elapsed on matrix multiplication of 1024x2048 . 2048x2048 on CPU: 50682.472656 ms.

     all results are correct!!!, speedup = 1224.202026
```

```
!./mat
```

```
     please type in m n and k
     100 100 100
     Time elapsed on matrix multiplication of 100x100 . 100x100 on GPU: 0.534784 ms.

     Time elapsed on matrix multiplication of 100x100 . 100x100 on CPU: 4.977536 ms.

     all results are correct!!!, speedup = 9.307564
```