CUDA Addition:

```
#include <math.h>
#include <time.h>
#include <iostream>
#include "cuda_runtime.h"


void cpuSum(int* A, int* B, int* C, int N){
   for (int i=0; i<N; ++i){
      C[i] = A[i] + B[i];
   }
}

__global__ void kernel(int* A, int* B, int* C, int N){
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   if (i < N){
      C[i] = A[i] + B[i];
   }
}

void gpuSum(int* A, int* B, int* C, int N){

   int threadsPerBlock = min(1024, N);
   int blocksPerGrid = ceil(double(N) / double(threadsPerBlock));

   kernel<<<blocksPerGrid, threadsPerBlock>>>(A, B, C, N);

}
```

```
bool isVectorEqual(int* A, int* B, int N){
    for (int i=0; i<N; ++i){
        if (A[i] != B[i])   return false;
    }
    return true;
}
int main(){
    int N = 2e7;
    int *A, *B, *C, *D, *d_A, *d_B, *d_C;
    int size = N * sizeof(int);

    A = (int*)malloc(size);
    B = (int*)malloc(size);
    C = (int*)malloc(size);
    D = (int*)malloc(size);



    for (int i=0; i<N; ++i){
        A[i] = rand() % 1000;
        B[i] = rand() % 1000;
    }



    // CPU
    clock_t start, end;

    start = clock();
    cpuSum(A, B, C, N);
    end = clock();

    float timeTakenCPU = ((float)(end - start)) / CLOCKS_PER_SEC;}
```

```
// GPU
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);
cudaMalloc(&d_C, size);

cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

start = clock();
gpuSum(d_A, d_B, d_C, N);
cudaDeviceSynchronize();
cudaMemcpy(D, d_C, size, cudaMemcpyDeviceToHost);

end = clock();
float timeTakenGPU = ((float)(end - start)) / CLOCKS_PER_SEC;

// free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

// Verify result
bool success = isVectorEqual(C, D, N);

printf("CPU Time: %f \n", timeTakenCPU);
printf("GPU Time: %f \n", timeTakenGPU);
printf("Speed Up: %f \n", timeTakenCPU/timeTakenGPU);
printf("Verification: %s \n", success ? "true" : "false");
```

Output:

```
CPU Time: 0.000901
GPU Time: 0.000089
Speed Up: 10.123595
Verification: true
```

CUDA Matrix Multiplication:

```
#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#define BLOCK_SIZE 16

__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;
    }
}
```

```
__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;
    }
}



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;
        }
    }
}
```

```
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();
    cudaDeviceSynchronize();
    // 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");
    }

    // 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);
    cudaFreeHost(h_cc);
    return 0;
}
```

Output:

```
Time elapsed on matrix multiplication of 80x80 . 80x80 on GPU: 0.179680 ms.

Time elapsed on matrix multiplication of 80x80 . 80x80 on CPU: 1.385056 ms.

all results are correct!!!, speedup = 7.708459
```