LAB 4
Name: Liam Lefferts
Write a program in CUDA to perform Matrix Addition.
1. Use the Profiler to report the times for

Matrix A and B Size	Threadsperblock 256	512	1024
A.512*512	1.82ms	2.05ms	2.60ms
B. 1024*1024	7.15ms	11.07ms	6.94ms
C. 8192*8192	1990.81 ms	3701.78ms	1776.31ms

Answer the following questions:
a. If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index

1. blockIdx.x+threadIdx.x
2. [✓ blockIdx.x*blockDim.x+threadIdx.x]
3. blockDim.x*threadIdx.y+threadIdx.y
4. blockIdx.x+lockDim.x

b. We want to use each thread to calculate two (adjacent) elements of a vector addition. Assume that variable I should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index?

1. blockIdx.x*blockDim.x+threadIdx.x +2
2. [✓ (blockIdx.x*blockDim.x+threadIdx.x)*2]
3. blockDim.x+threadIdx.x
4. blockIdx.x*2*blockDim.x+threadIdx.x

c. For a vector addition, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid

1. 2000
2. 2024
3. [✓ 2048 [(512*4) = 2048 threads - 2000 working threads = 48 canceled threads]]
4. 2096

```
   =======================================Appendix======================================
#include<stdlib.h>
#include<stdio.h>
#include <cuda_runtime.h>

#define TILE_WIDTH 16 //or 33 for 1024 threadsperblock
#define seed 13

//kernel function
__global__ void matrixMul(float *dev_A, float *dev_B, float *dev_C, int matrixWitdh)
{
   // allocate tiles in __shared__ memory
   __shared__ float A_tile[TILE_WIDTH][TILE_WIDTH];
   __shared__ float B_tile[TILE_WIDTH][TILE_WIDTH];

   int bx = blockIdx.x;
   int by = blockIdx.y;

   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // calculate the row & col index to identify element to work on
   int row = by * TILE_WIDTH + ty;
   int col = bx * TILE_WIDTH + tx;

   float partial = 0.0;
   int m;

// loop over the tiles of the input in phases
   for( m=0 ; m < matrixWitdh/TILE_WIDTH; m++) {

// collaboratively load tiles into __shared__
      A_tile[ty][tx] = dev_A[row * matrixWitdh + (m * TILE_WIDTH + tx)];
      B_tile[ty][tx] = dev_B[col + (m * TILE_WIDTH + ty) * matrixWitdh];

      __syncthreads();
      int k;
// dot product between row of A_tile and col of B_tile
      for(k=0; k< TILE_WIDTH; k++)
         partial += A_tile[ty][k] * B_tile[k][tx];
// synchronize threads before adding the values on the cache
      __syncthreads();
//put the results in the result matrix
      dev_C[row * matrixWitdh + col] = partial;
   }
}
```

```
int main(int argc, char **argv) {
   srand(seed);

   if(argc != 2) {
      printf("Usage /lab4_4 <matrixWitdh>");
      return 1;
   }
   int matrixWitdh = atoi(argv[1]);

//allocate space for host matrices
   float *h_A = (float*) malloc(matrixWitdh * matrixWitdh * sizeof(float));
   float *h_B = (float*) malloc(matrixWitdh * matrixWitdh * sizeof(float));
   float *h_C = (float*) malloc(matrixWitdh * matrixWitdh * sizeof(float));

//Matrix Fill
   int i,j;
   for(i=0; i<matrixWitdh; i++) {
      for(j=0; j<matrixWitdh; j++) {
         h_A[i * matrixWitdh + j] = (float)rand()/((float)RAND_MAX/10.0);
         h_B[i * matrixWitdh + j] = (float)rand()/((float)RAND_MAX/10.0);
      }
   }

//cuda alloate the device  matrices
   float *d_A, *d_B, *d_C;
   cudaMalloc((void**) &d_A, matrixWitdh * matrixWitdh * sizeof(float));
   cudaMalloc((void**) &d_B, matrixWitdh * matrixWitdh * sizeof(float));
   cudaMalloc((void**) &d_C, matrixWitdh * matrixWitdh * sizeof(float));

//define grid and block dimentions
   dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
   dim3 dimGrid(matrixWitdh/TILE_WIDTH, matrixWitdh/TILE_WIDTH, 1);

   float elapsedTime;
   cudaEvent_t start, stop;
   cudaEventCreate(&start);
   cudaEventCreate(&stop);
   cudaEventRecord(start, 0);

//copy result from device to host
   cudaMemcpy(d_A, h_A, matrixWitdh * matrixWitdh * sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, matrixWitdh * matrixWitdh * sizeof(float), cudaMemcpyHostToDevice);
//call Kernel function
   matrixMul<<< dimGrid, dimBlock >>>(d_A, d_B, d_C, matrixWitdh);
   cudaMemcpy(h_C, d_C, matrixWitdh* matrixWitdh * sizeof(float), cudaMemcpyDeviceToHost);
   cudaEventRecord(stop, 0);
   cudaEventSynchronize(stop);
   cudaEventElapsedTime(&elapsedTime, start, stop);
```

```
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    printf("For tiled version, the elapsed time is %.4f(ms).\n", elapsedTime);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```