CUDA Threads

1) Complete and run the following `vecAdd4.cu` program, and see if it works for any $n > T$, e.g. $n = 1234$ and $T = 64$.

```
#define n 1024      // size of vectors
#define T 240        // number of threads per block

__global__ void vecAdd(int *A, int *B, int *C) {

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

    if (i < n)                  // allows for more threads than vector elements
        C[i] = A[i] + B[i];     // some unused
}

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

   int blocks = (n + T - 1) / T;    // efficient way of rounding to next integer
    …
    vecAdd<<<blocks, T>>>(devA, devB, devC);
    …

}
```

2) Create a CUDA program to fill in the array A[256] using 4 thread blocks. Each block has 64 threads. Each element A[i] = i as shown below.

0	1	2	3	4	5	6	…..	253	254	255

3) Create and run the `matmul2.cu` program which multiplies two square matrices. *Width* is strictly a multiple of *TILE_WIDTH*.

```
#include <stdio.h>

#define Width 32      // size of Width x Width matrix
#define TILE_WIDTH 16

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int ncols) {

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

    // Pvalue is used to store the element of the output matrix
    // that is computed by the thread

    float Pvalue = 0;
    for (int k = 0; k < ncols; ++k) {
       float Melement = Md[row*ncols+k];
       float Nelement = Nd[k*ncols+col];
       Pvalue += Melement * Nelement;
    }

    Pd[row*ncols+col] = Pvalue;
}
```

```
int main (int argc, char *argv[] ) {

    int i,j;
    int size = Width * Width * sizeof(float);
    float M[Width][Width],N[Width][Width],P[Width][Width];
    float* Md, *Nd, *Pd;

    for (i=0; i < Width; i++) {
        for (j=0; j < Width; j++) {
           M[i][j] = 1; N[i][j] = 2;
        }
    }

    cudaMalloc( (void**)&Md, size);
    cudaMalloc( (void**)&Nd, size);
    cudaMalloc( (void**)&Pd, size);

    cudaMemcpy( Md, M, size, cudaMemcpyHostToDevice);
    cudaMemcpy( Nd, N, size, cudaMemcpyHostToDevice);

    // Setup the execution configuration
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
    dim3 dimGrid(Width/TILE_WIDTH,Width/TILE_WIDTH);

    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);

    // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);

    for (i=0; i < Width; i++) {
       for (j=0; j < Width; j++) {
           printf("%.2f ",P[i][j]);
       }
       printf("\n");
    }

}
```

4) Modify the program `matmul2.cu` into `matmul3.cu` to work with square matrices of which the matrix *Width* not necessary a multiple of *TILE_WIDTH*. For example, Width = 32 but TILE_WIDTH=5.