CUDA Threads

1. Complete and run the following `vecAdd4.cu` program, and see if it works for any $n > T$, for example, when $n = 1234$ 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 the next integer

    ...  // see the old code
    vecAdd<<<blocks, T>>>(devA, devB, devC);
    ...
}
```

Figure 1: vecAdd4.cu

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

1	2	3	4	5	6	$\cdots$	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;
}

```

Figure 2: mutmal2.cu (part 1/2)

```
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 thread!
    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");
    }
}
```

Figure 3: mutmal2.cu (part 2/2)

4. Modify the program `matmul2.cu` into `matmul3.cu` to work with square matrices of arbitrary size. (`Width` not neccessary a multiple of `TILE_WIDTH`)

5. Create a MS Word document (u5xxxxxx.docx), and put your source code from step 1. to step 4., along with the screeshots of their result.

    - `vecAdd4.cu`
    - `vecFill.cu`
    - `matmul2.cu`
    - `matmul3.cu`