ECE 1782

CUDA Programming Assignment

Due January 31, 2019

1. Specification

Write a CUDA program that adds two matrices of the same size containing elements of the type `float`.

Your CUDA program should accept two arguments:

1. An integer that specifies the number of columns in the matrices.
2. An integer that specifies the number of rows in the matrices.

Each time the CUDA program is invoked, it will invoke a GPU kernel, here called `f_matadd()`, exactly once.

Your CUDA program should output 4 numbers on one line terminated with the newline character and then exit:

```
<total_time> <CPU_GPU_transfer_time> <kernel_time> <GPU_CPU_transfer_time><nl>
```

Each of these numbers are defined as follows:

- `<total_time>`: the time in seconds from the point just before data is transferred to the GPU to just after the result data is transferred back from the GPU (`timeStampD–timeStampA` in the code below).
- `<CPU_GPU_transfer_time>`: the time in seconds it takes to transfer the two matrices to the GPU (`timeStampB`–timeStampA in the code below).
- `<kernel_time>`: the time in seconds it takes the GPU to execute the kernel (`timeStampC–timeStampB` in the code below).
- `<GPU_CPU_transfer_time>`: the time in seconds it takes the GPU to transfer the result matrix back to the CPU (`timeStampD–timeStampC` in the code below).

The numbers should be output with 6 digits of precision (“`%.6f`”) and be separated by white space. If an error occurs, then your program should output one line starting with “`Error: `” followed by a description of the error before exiting.

The two matrices that will be added, here called `A` and `B`, should be initialized host-side (before copying them to GPU global memory) as follows:

- `A[i,j] = (float)  (i+j)/3.0 ;`
- `B[i,j] = (float) 3.14*(i+j) ;`

The result matrix produced on the GPU, here called d_C, should be copied back to CPU memory and (to check correctness) should be compared against the result of a corresponding matrix addition computed CPU side.

The details on how to submit your code will be provided at a later time. However, when submitting, your entire CUDA program should be contained in a source file named `<your_student_no>.cu`.

When running your experiments, you will notice that transferring the matrices over the PCIe bus dominates the total time. This is because matrix addition is not very compute intensive. That is OK, as this is just an exercise for you to familiarize yourself with CUDA programming and for getting a feel for some of the things that affect performance. Optionally, you may want to also run versions of the above code (but not submit) where matrices are of type `int` and `double` (instead of type `float`) to see what difference it makes to your measured execution times.

2. Possible rough program sketch

Here is one possible program structure:

```
// time stamp function in seconds
double getTimeStamp() {
    struct timeval  tv ;
    gettimeofday( &tv, NULL ) ;
    return (double) tv.tv_usec/1000000 + tv.tv_sec ;
}

// host side matrix addition
h_addmat(float *A, float *B, float *C, int nx, int ny){ … }

// device-side matrix addition
__global__ f_addmat( float *A, float *B, float *C, int nx, int ny ){
    // kernel code might look something like this
    // but you may want to pad the matrices and index into them accordingly
    int ix = threadId.x + bloackId.x*blockDim.x ;
    int iy = threadId.y + bloackId.y*blockDim.y ;
    int idx = iy*ny + ix ;
    if( (ix<nx) && (iy<ny) )
        C[idx] = A[idx] + B[idx] ;
}

int main( int argc, char *argv[] ) {
    // get program arguments
    if( argc != 3) {
        printf( “Error: wrong number of args\n” ) ;
        exit() ;
    }
    int nx = atoi( argv[2] ) ; // should check validity
    int ny = atoi( argv[3] ) ; // should check validity
    int noElems = nx*ny ;
    int bytes = noElems * sizeof(float) ;
    // but you may want to pad the matrices…
```

```
    // alloc memory host-side
    float *h_A = (float *) malloc( bytes ) ;
    float *h_B = (float *) malloc( bytes ) ;
    float *h_hC = (float *) malloc( bytes ) ; // host result
    float *h_dC = (float *) malloc( bytes ) ; // gpu result

    // init matrices with random data
    initData( h_A, noElems ) ; initData( h_B, noElems ) ;

    // alloc memory dev-side
    float *d_A, *d_B, *d_C ;
    cudaMalloc( (void **) &d_A, bytes ) ;
    cudaMalloc( (void **) &d_B, bytes ) ;
    cudaMalloc( (void **) &d_C, bytes ) ;

    double timeStampA = getTimeStamp() ;

    //transfer data to dev
    cudaMemcpy( d_A, h_A, bytes, cudaMemCpyHostToDevice ) ;
    cudaMemcpy( d_B, h_B, bytes, cudaMemCpyHostToDevice ) ;
    // note that the transfers would be twice as fast if h_A and h_B
    // matrices are pinned

    double timeStampB = getTimeStamp() ;

    // invoke Kernel
    dim3 block( 32, 32 ) ;   // you will want to configure this
    dim3 grid( (nx + block.x-1)/block.x, (ny + block.y-1)/block.y ) ;

    f_addmat<<<grid, block>>>( d_A, d_B, d_C, nx, ny ) ;

    cudaDeviceSynchronize() ;

    double timeStampC = getTimeStamp() ;

    //copy data back
    cudaMemCpy( h_dC, d_C, bytes, cudaMemCpyDeviceToHost ) ;

    double timeStampD = getTimeStamp() ;

    // free GPU resources
    cudaFree( d_A ) ; cudaFree( d_B ) ; cudaFree( d_C ) ;
    cudaDeviceReset() ;

    // check result
    h_addmat( h_A, h_B, h_hC, nx, ny ) ;
    // h_dC == h+hC???

   // print out results
}
```