Homework #5 Solution Set

1. **1D Convolution - code (35 points)**

   (a) *Solution:*
   `conv1h_basic()`: In addition to making the call to the CUDA kernel, this function takes care of all the usual CUDA memory allocation and copying. Note the call to `cudaMemcpyToSymbol` that copies the stencil into constant memory. We are using a 1D block of threads, since we are performing the convolution using a 1D stencil. We use a 2D grid of blocks, where the y-dimension is `height`, since our blocks are 1D, and the x-dimension is calculated to contain enough blocks to cover a full row of pixels of the image.
   `conv1h_basic_kernel()`: Our job is to calculate the value of a single pixel by applying the convolution stencil to the pixels around it. `x` and `y` will represent the coordinates of the pixel we are calculating (note that, since we are using 1D blocks, `blockIdx.y` will be 0, and hence the abbreviated declaration of `y`). We just loop over the stencil, making sure to not include values that fall outside the current row of the image. `cur_x` represents where in the image array the stencil is currently being applied, while `image_offset` represents the corresponding pixel of the image. So we check to make sure that we are within the bounds of the current row of the image (`cur_x >= 0 && cur_x < width`), and if so, we multiply the value of the pixel with that of the stencil, and add the result to the `sum` variable. Finally, we write the value of `sum` into the pixel at coordinates `x,y`.

   (b) *Solution:*
   `conv1v_basic()`: As above, this function does all the CUDA memory setup and then calls the kernel. Again, we use a 1D block of threads, and a 2D grid of blocks. Whereas above we had one block for each row of the image, here we have one block for each *column* of the image (i.e., the grid's y-dimension is `width`). Again, the x-dimension is calculated to contain enough blocks to cover a full column of pixels of the image.
   `conv1v_basic_kernel()`: The code structure is identical to `conv1h_basic_kernel()` above. Here, we use the clever trick of swapping the `x` and `y` variables. Now we can use `y` to represent the current column, and `x` to represent the current row. Again, `cur_y` represents where in the image array the stencil is currently being applied and `image_offset` represents the corresponding pixel of the image. Note that the index calculation for `image_offset` is done differently from above, due to the swapping of `x` and `y`, but it still follows the common 2D indexing pattern of "`y * width + x`".

   (c) *Solution:*
   `conv1to2_basic()`: This function should be easy once you have `conv1h_basic_kernel()` and `conv1v_basic_kernel()` working. We basically just combine the code in the two functions `conv1h_basic()` and `conv1v_basic()`. We do all of the memory allocations and set up two thread grids, `gridDim_h` and `gridDim_v`, using the same parameters as in (a) and (b). We then simply make sequential calls to `conv1h_basic_kernel()` and `conv1v_basic_kernel()`. To avoid unnecessary memory allocations, we use the input array to `conv1h_basic_kernel()` as the output array for `conv1v_basic_kernel()`.

2. **1D Convolution - performance (20 points)**

   (a) *Solution:*
   Here is a simple, naive (slightly incorrect) approach: For the horizontal stencil, there are a total of $h \cdot w$ pixel values to calculate. For each such pixel we loop over the stencil and perform one multiplication and one addition. Thus, for the horizontal stencil we perform $2hws_x$ operations. Similarly, the vertical stencil we perform $2hws_y$ operations, for a total of $2hw(s_x + s_y)$ operations.

(b) *Solution:*
Again, here is a naive approach: For the horizontal kernel, for each of the $h \cdot w$ pixels, and for each of the s_x elements of the stencil, we read a pixel of the input from global memory. That's a total of s_x memory reads for each pixel of the input image. Then, we need to write the computed value back to global memory. That's a total of $hw(s_x + 1)$ global memory accesses. Similarly, the vertical kerne makes $hw(s_y + 1)$ accesses, for a total of $hw(s_x + s_y + 2)$

(c) *Solution:*
CGMA is calculated as $\frac{\textit{Number of operations for computation}}{\textit{Number of global memory accesses}}$. Thus, CGMA is just the ratio of the results from the previous two questions. That is $\frac{2hw(s_x+s_y)}{hw(s_x+s_y+2)} = \frac{2(s_x+s_y)}{s_x+s_y+2}$

(d) *Solution:*
Throughput is being measured in pixels/second. The `booby.ppm` image is 1071×1850, so has 1981350 pixels. Thus, if T is the time in seconds to execute `conv1to2_basic()`, the throughput would be $\frac{1981350}{T}$ pixels/second.

3. **1D Convolution with Tiling (45 points)**

(a) *Solution:*
`conv1h_tiled_kernel()`: As with tiled versions of other functions we've seen, the kernel computation proceeds in two stages. In the first stage, data is collaboratively loaded by each thread into the block's shared memory. In the second stage, threads use the data stored in shared memory to compute the convolution for pixels they are responsible for.

Since the current block needs data that falls outside the portion of the array it is responsible for calculating, we need to also load a "halo" into shared memory. `halo_left_start` and `halo_right_end` represent the boundaries of the entire portion of the array to be loaded into memory.

Then in the first loop of the kernel, the current thread iterates over the array from `halo_left_start` to `halo_right_end`, checks that the current index falls within the current row of the image, and then loads the value of the pixel it finds there into `s_inputs` in shared memory. It takes steps of size `BLOCK_SIZE_X` so that each block loads a contiguous portion of the image at a time.

After this first loop, the current thread has loaded all its data into `s_inputs`. We have to wait at the `_syncthreads()` before continuing, to make sure that all the other threads in this block have also finished loading their values. At this point, calculating the value of the convolution for a single pixel is done in a manner completely analogous to the non-tiled version, except now the data is read from shared memory.

`conv1v_tiled_kernel()`: We follow exactly the same pattern as with `conv1h_tiled_kernel()`, and make the same indexing alterations as we made when going from `conv1h_basic()` to `conv1v_basic()`.

`conv1to2_tiled`: As with `conv1to2_basic`, we just need to combine the code from `conv1h_tiled` and `conv1v_tiled` into a single function. As before, we have two `blockDim` and `gridDim` structs, one for each kernel call, and we use the input array to `conv1h_tiled_kernel()` as the output array for `conv1v_tiled_kernel()`.

(b) *Solution:*
Assume the tile and its corresponding block are 1D in this subquestion since there is a trivial mapping between K&H and the implementation in this subquestion.

First, let's compute the global memory accesses.

(1) Consider moving elements from the global memory into the shared memory within one block in `conv1h_tiled_kernel()`. K&H figure 8.10 (in the second edition) or 7.10 (in the third edition) is a good illustration.
Three types of elements are move: left halo element, right halo element and middle elements. The number of elements belonging to the two halos is $s_x - 1$ in total. The number of elements in the middle is the size of the tile t. For one tile (or one block), there are $s_x + t - 1$ global memory accesses to move elements into the shared memory.

These elements in shared memory are used to compute t results in the middle part. There are t global memory accesses to write results back.
Therefore, the number of global memory accesses in a tile is $s_x + 2t - 1$.

(2) Consider tiles (or blocks) in one image row.
There are $\lceil \frac{w}{t} \rceil$ tiles. To simplify the result, we don't subtract elements outside the image boundary and the total global memory accesses in a row is $\lceil \frac{w}{t} \rceil (s_x + 2t - 1)$.

(3) Consider tiles (or blocks) in the whole image.
Since the stencil is 1D in this case, computations in different rows are independent to each other. It is equivalent to run h 1D tiled convolutions on each row in parallel. Thus, the overall global memory accesses is $h\lceil \frac{w}{t} \rceil (s_x + 2t - 1)$.

Now, let's consider floating point operations in the similar per-tile fashion.

In each tile, there are t pixels in the middle and similar to Q2(a), each one requires $2s_x$ computations. So there are $2s_x \cdot t$ operations per tile. In each row, the number of operations is $2\lceil \frac{w}{t} \rceil s_x \cdot t$. Therefore, the number of operations in the whole image is $2h\lceil \frac{w}{t} \rceil s_x \cdot t$.

By replacing s_x with s_y and swapping w and h, you should get the result for the horizontal direction.

The CGMA is

$$\frac{2h\lceil \frac{w}{t} \rceil s_x \cdot t + 2w\lceil \frac{h}{t} \rceil s_y \cdot t}{h\lceil \frac{w}{t} \rceil (s_x + 2t - 1) + w\lceil \frac{h}{t} \rceil (s_y + 2t - 1)}.$$

(c) *Solution:*
The throughput calculation is the same as for Question 2 (d).

4. **cuBLAS (40 points)**

(a) *Solution:*
The three BLAS subroutines are (1) `cublasSgemv`, (2) `cublasSnrm2`, (3) `cublasSscal`.

(b) *Solution:*
See `hw5_blas.cu`.

(c) *Solution:*
Measured on `lin21`, the runtime of the power iteration computation is `8.800e-02 s` and the runtime including the memory operations (`cudaMallc`, `cudaFree` and memory movement between CPU and GPU) is `1.760e-01 s`.

(d) *Solution:*
For this subquestion, we only consider the power iteration computation time (`8.800e-02 s`). If the kernel uses this runtime only for global memory accesses only, the number of single precision floating point values it touches is

$$\frac{98.2 \times 2^{30} \times 0.088}{4} = 2.32 \times 10^9.$$

(e) *Solution:*
Given A is a $n \times n$ matrix and b is a n dimensional vector, consider a single iteration.

(1) $b \leftarrow Ab$.
For each element in the resulting b, the total number of additions and multiplications is $2n-1$ and we round it to $2n$. Thus, the total number is roughly $2n^2$.

(2) $\lambda \leftarrow ||b||$.
Given the 2-norm is defined as $\sqrt{\sum_{i=1}^{n} b_i^2}$, the total number of additions and multiplications is rounded to $2n$. There is one square root operation per iteration. Since the runtime is dominated by the $2n$ additions and multiplications, we omit the single square root operation.

(3) $b \leftarrow (\frac{1}{\lambda})b$.
There are n multiplications.

The numbers above sum up to $2n^2 + 3n$ per iteration. Running these steps for k iterations gives $k(2n^2 + 3n)$ floating point operations. Plugging in $k = 500, n = 2048$ gives 4197376000.

(f) *Solution:*
The kernel uses the runtime for not only global memory accesses but also computations. However, since the actual code is hidden inside BLAS library, we can only obtain a low bound on CGMA by assuming the kernel uses all the runtime for global memory accesses. Then, CGMA is lower bounded by

$$\frac{4197376000}{2.32 \times 10^9} = 1.81.$$

Note (added by Mark): This problem is built on BLAS level-2 functions. In particular, it performs repeated matrix-vector multiply. Each matrix-vector product is performed by a separate kernel. This means that the matrix must be loaded from global memory for each product. Each matrix element is used in **one** multiply-add when performing a matrix-vector product. Thus we get one multiply, and one add per memory read, just for the matrix. This would give a CGMA of 2. The measured 1.8 number suggests that the cuBLAS library gets pretty close to this limit. It also says that the performance of the cuBLAS functions for this problem is limited by the GPU's memory bandwidth – the huge floating-point throughput of the GPU isn't being exercised. Kind of like SAXPY.

Reading the matrix involves many more memory operations than reading the vector, or storing the final result. We can surmise that the cuBLAS function loads the vector into on-chip memory because each element of an N-element vector is accessed N times when multiplying the vector by an N-by-N matrix, and I'm assuming that the cuBLAS programmers write better code that I do. This suggests that the cuBLAS code performs N^2 memory reads to access the matrix, and N reads to load the vector, and N writes to write the result. The N^2 memory accesses for the matrix dominate the $2N$ accesses for the vector. This justifies the focus on the accessing the matrix elements in the previous paragraph.