DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Parallel Computer Architecture ECEC-622

TITLE: CUDA Assignment 3: 2D Convolution

GROUP MEMBERS: Gregory Matthews and Mark Klobukov

INSTRUCTOR: Dr. Kandasamy

DATE SUBMITTED: 3/16/2017

DATE DUE: 3/19/2017

Assignment description:

This week's assignment calls to complete the functionality of a matrix by computing the 2D matrix convolution of it on the GPU. The standard convolution formula will incorporate a 5x5 convolution kernel to compute the convolution of Matrix A and Matrix B. The convolution formula can be seen in the given equation:

$$C(i,j) = \sum_{m=0}^{4} \sum_{n=0}^{4} A[m][n] * B[i+m-2][j+n-2]$$

This equation assumes that index i is from 0 to the height of Matrix B, and index j is from 0 to the width of Matrix B. m and n are the height and width indices for the convolution kernel, respectively. All other elements that fall outside of the boundary shall be padded with zeros, or entirely skipped over via a conditional statement.

Parallelization Approach:

Because there's no cache coherence issues in the convolution formula, this makes 2D convolution highly parallelizable as a result. For our approach to parallelizing 2D convolution, each thread gets a particular index of Matrix B within the thread's current thread block, and computes the convolution of that element with the entire convolution kernel Matrix A. This is done by computing the partial sum of each operation, and then storing the total sum into the respective Matrix C index for that thread. In the global memory approach, the execution can be solved relatively simply as a result of halo elements not being a problem because each thread will simply access Matrix B from global memory. The global memory approach however, will have to worry about ghost elements, which can be done by padding 0's or a conditional statement to check Matrix boundaries.

For the optimized approach, Matrix B can be stored in shared memory to reduce the transfer time from CPU to GPU and improve performance. Since each thread block only needs access to a thread block's worth of shared memory of B to compute the convolution, we can keep the shared memory size small to thread_block_size*thread_block_size. The only thing to worry about are the halo elements, which are not stored in the current shared memory for Matrix B at the current thread, but are stored in the previous thread and the subsequent thread. We can take advantage of the L2 cache and hope that these values were already cached by a previously executed thread block, and if it's not found, we simply can check global memory for the value. Each thread will compute partial sums for each multiplication operation on the convolution

kernel, and transfer the total sum to the corresponding output Matrix C's address. The convolution kernel Matrix A can be stored into constant memory by using cudaMemCpyToSymbol(). Because there'll be no changes made to the convolution kernel when running the program, the Matrix variable can be kept in constant memory. Nvidia hardware provides us with 64 KB of storage for constant memory variables that it treats differently than global memory, and reduces the overall required memory bandwidth. The cost of reading from constant memory is only one read from device memory on a cache miss, otherwise it's also one read from constant cache. This will greatly improve the performance when threads initiate reads.

CUDA code explanation:

1.) Host Function

On the CPU side of the program, the code shown below begins by allocating memory for the 3 Matrices that'll be sent to the GPU after initiating kernel execution. Matrix M and the convolution kernel matrix N is copied over into the GPU's variable addresses for the global memory approach. The thread_block is created by taking the total number of elements and dividing it by the thread_block_size, which is a macro defined in the header file. The grid will consist thread_block_size in the x and y direction because we'll be working with Matrices.

```
void ConvolutionOnDevice(const Matrix M, const Matrix N, Matrix P_global, Matrix P_shared)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd_global = AllocateDeviceMatrix(P_global);
    CopyToDeviceMatrix(Pd_global, P_global); // Clear memory

    Matrix Pd_shared = AllocateDeviceMatrix(P_shared);
    CopyToDeviceMatrix(Pd_shared, P_shared); // Clear memory

    // Setup the execution configuration
    int num_elements = N.width;
    dim3 thread_block(THREAD_BLOCK_SIZE, THREAD_BLOCK_SIZE, 1);
    int num_thread_blocks = ceil((float)num_elements/(float)THREAD_BLOCK_SIZE);
    dim3 grid(num_thread_blocks, num_thread_blocks, 1);
```

Once all variables have been allocated and copied into the device sides variables, the kernel is then executed to give control over to the GPU to compute the convolution formula.

```
// Launch the device computation threads!
struct timeval start, stop;

gettimeofday(&start, NULL);
ConvolutionKernel<<<grid, thread_block>>>(Md, Nd, Pd_global);
cudaThreadSynchronize();
gettimeofday(&stop, NULL);

printf("Global Memory Time = %fs. \n", (float)(stop.tv_sec \
 -start.tv_sec +(stop.tv_usec - start.tv_usec)/(float)1000000));

 // Read P from the device
CopyFromDeviceMatrix(P_global, Pd_global);
FreeDeviceMatrix(&Pd_global);
check_for_error("KERNEL FAILURE");
```

For the optimized approach, cudaMemCpyToSymbol() is used to store the convolution kernel matrix into constant memory on the GPU. Constant memory is used for the convolution kernel because this Matrix will be used specifically for reading only, therefore memory bandwidth can be implemented by not storing the Matrix into global memory. The kernel is then invoked to execute the convolution formula on the device side by using the optimized CUDA approach.

```
// We copy the mask to GPU constant memory in an attempt to improve the performance
cudaMemcpyToSymbol(kernel_c, M.elements, KERNEL_SIZE*KERNEL_SIZE*sizeof(float));

// Launch the device computation threads!
gettimeofday(&start, NULL);
ConvolutionKernel_optimized<<<grid, thread_block>>>(Nd, Pd_shared);
cudaThreadSynchronize();
gettimeofday(&stop, NULL);
printf("Optimized CUDA time = %fs. \n", (float)(stop.tv_sec \
-start.tv_sec +(stop.tv_usec - start.tv_usec)/(float)1000000));

// Read P from the device
CopyFromDeviceMatrix(P_shared, Pd_shared);

FreeDeviceMatrix(&Md);
FreeDeviceMatrix(&Nd);
FreeDeviceMatrix(&Pd_shared);
```

2.) Device Function

For the naive Global Memory approach, all Matrices are stored into Global Memory, making thread read operations relatively simple, at the expense of performance and optimization. The row and column indices x and y, respectively, are calculated for each thread in a thread block, and initializing the Matrix N's height and width into variables.

A base case conditional statement will check that the indices i and j have not fallen out of the bounds of the Matrix dimensions before performing the convolution computation. For each thread that now has an element to compute on the output Matrix P, the convolution kernel beginning and ending indices are computed for each thread since it will need these indices to multiply its surrounding elements by all the elements in the mask. The final nested for loop is the actual convolution computation and involves computing the partial sum of each N index by the mask M index, which are both stored in global memory, therefore halo elements are not even considered here. The final sum for each thread is then loaded into the output Matrix P. Once all thread blocks have successfully completed their runs, the convolution of the entire matrix is then completely solved.

```
__global__ void ConvolutionKernel(Matrix M, Matrix N, Matrix P)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int wN = N.width;
    int hN = N.height;

    if (i >= N.height || j >= N.width) return;

    // For each element in the result  matrix
    double sum = 0;
    // check the start and end values of m and n to prevent
    // overrunning the matrix edges
    unsigned int mbegin = (i < 2)? 2 - i : 0;
    unsigned int mend = (i > (hN - 3))?
                            hN - i + 2 : 5;
    unsigned int nbegin = (j < 2)? 2 - j : 0;
    unsigned int nend = (j > (wN - 3))?
                            (wN-j) + 2 : 5;

    // overlay M over N centered at element (i,j).  For each
    //  overlapping element, multiply the two and accumulate
    for(unsigned int m = mbegin; m < mend; ++m) {
        for(unsigned int n = nbegin; n < nend; n++) {
            sum += M.elements[m * 5 + n] *
                    N.elements[wN*(i + m - 2) + (j+n - 2)];
        }
    }
    // store the result
    P.elements[i*wN + j] = (float)sum;

}
```

For the optimized approach, we remove global memory access for Matrices M and N, and replace them with shared and constant memory, respectively. The shared memory variable N is allocated into device memory as N_ds in the code shown below, and is made to the size of thread_block_size * thread_block_size. The shared variable is then loaded locally for each thread_block to use in its convolution computation. The starting and next starting x and y indices for each thread are calculated, as well as the starting x and y indices for N. The nested for loop does the actual convolution computation by incrementing over the predefined variable KERNEL_SIZE which is 5 for both for loops, which allows for complete iteration over the convolution kernel matrix indices, which each thread will need to compute its corresponding element's convolution equation. Inside the outer for loop, we create a variable N_index_y, which computes the y index that the current thread needs to index N to compute a partial sum of the convolution. Inside the next for loop, another variable N_index_x is created which computed the x index that the current thread needs to index N. The following conditional statement that follows, checks that the current thread block, and the next thread block is within the bounds of the N indices for that given thread, and if it is, it uses the shared memory resource to access N_ds and compute a partial sum of the convolution. If it isn't stored in the shared memory resource, we then access global memory to find the value indexed in N. The global memory call will implicitly check the L2 cache before going to global memory, and if a previous thread block has already cached that particular N index, then the current thread block can then access it without resorting to accessing global memory. The total partial sum is then loaded into Matrix P.

```
__global__ void ConvolutionKernel_optimized(Matrix N, Matrix P) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;

    __shared__ float N_ds[THREAD_BLOCK_SIZE*THREAD_BLOCK_SIZE];
    N_ds[THREAD_BLOCK_SIZE * threadIdx.y + threadIdx.x] = N.elements[N.width * y + x];
    __syncthreads();

    int this_start_x = blockIdx.x*blockDim.x;
    int this_start_y = blockIdx.y*blockDim.y;
    int next_start_x = (blockIdx.x+1)*blockDim.x;
    int next_start_y = (blockIdx.y+1)*blockDim.y;

    int N_start_x = x - (KERNEL_SIZE/2);
    int N_start_y = y - (KERNEL_SIZE/2);

    double Pvalue = 0;

    for (int k = 0; k < KERNEL_SIZE; k++) {
        int N_index_y = N_start_y + k;
        if (N_index_y >= 0 && N_index_y < N.width) {
            for (int r = 0; r < KERNEL_SIZE; r++) {
                int N_index_x = N_start_x + r;
                if (N_index_x >= 0 && N_index_x < N.width) {
                    if ((N_index_x >= this_start_x)
                                && (N_index_y >= this_start_y)
                                && (N_index_x < next_start_x)
                                && (N_index_y < next_start_y) ) {
                Pvalue += N_ds[THREAD_BLOCK_SIZE * (threadIdx.y + k - KERNEL_SIZE/2) \
                + threadIdx.x + r - (KERNEL_SIZE/2)] * kernel_c[KERNEL_SIZE * k + r];
                    }
                    else {
                Pvalue += N.elements[N.width * N_index_y + N_index_x] \
                * kernel_c[KERNEL_SIZE * k + r];
                    }
                }
            }
        }
    }
    P.elements[P.width * y + x] = (float)Pvalue;
}
```

Results:

Multiple tests were run on the program, and the timing results are shown in the table below. For each matrix size, three different thread-block sizes were tested.

Matrix Size	Thread-block size	Time Serial (s)	Time naïve (s)	Time optimized (s)	Speedup Naïve	Speedup Optimized
512x512	8	0.018511	0.000402	0.000131	46.047	141.305
	16	0.018466	0.00046	0.000129	40.143	143.147
	32	0.018533	0.00058	0.000134	31.953	138.306
1024x1024	8	0.061675	0.001097	0.000463	56.222	133.207
	16	0.061594	0.001327	0.000462	46.416	133.320
	32	0.061562	0.001755	0.000465	35.078	132.391
2048x2048	8	0.202808	0.003839	0.001765	52.828	114.905
	16	0.21312	0.004747	0.001746	44.896	122.062
	32	0.198834	0.006477	0.001767	30.698	112.526

Note the last two columns of the table above. The speedup achieved with using constant and shared memory is higher than the speedup with just global memory by a factor of about 3.5. Sensitivity of the program performance to the thread-block size was also evaluated. The following table summarizes the tests for the naïve (global memory only) approach. The rightmost column shows the average increase in run-time per each added thread in a thread block. The data shows that the kernel performance is slightly decreased with a larger thread-block size. The run-time increase is insignificant, however.

Matrix Size	**Change in # threads in a block**	**Total change in Time (s)**	**Change in Time per Extra thread in a block (s)**
512x512	8->16	0.000058	0.00000725
	16->32	0.00012	0.0000075
1024x1024	8->16	0.00023	0.00002875
	16->32	0.000428	0.00002675
2048x2048	8->16	0.000908	0.0001135
	16->32	0.00173	0.000108125

Similar measurements were made for the optimized case (shared and constant memory). The table below summarizes our findings. This kernel has a much lower sensitivity to the thread-block size, compared to the global-memory-only approach. It appears that the performance peaks when the number of threads per block is 16. But as in the global-memory-only kernel, the sensitivity of program performance to the thread-block size is minimal.

Matrix Size	Change in # threads in a block	Total change in Time (s)	Change in Time per Extra thread in block (s)
512x512	8->16	-0.0000020	-0.0000002500
	16->32	0.0000050	0.0000003125
1024x1024	8->16	-0.0000010	-0.0000001250
	16->32	0.0000030	0.0000001875
2048x2048	8->16	-0.0000190	-0.0000023750
	16->32	0.0000210	0.0000013125

Using the GPU for parallel computations inevitably introduces some overhead to the program. The overhead is due to having to allocate memory on the GPU and transfer data from the host side to the device side before computation and back after computation. In addition, the allocated memory must be freed before the host function exits. The following timings were obtained for the overhead. The last two columns help understand the importance of using streams for host-device communication in order to improve the parallel code performance. Interestingly, overhead does not seem to scale up with the increasing matrix sizes.

Matrix Size	Overhead Naïve (s)	Overhead Optimized (s)	Fraction of Run Time Naïve	Fraction of Run Time Optimized
512x512	0.0268	0.02678	46.207	204.427
1024x1024	0.02651	0.02649	24.166	57.214
2048x2048	0.02525	0.02523	6.577	14.295

To calculate the GFLOPs for our code, we used the following formula:

$$GFLOPs = \frac{(\#elements\ in\ mask)\times(\#\ elements\ in\ input\ matrix)}{10^9}$$
$$= \frac{25\times(matrix\ dimension)^2}{10^9}$$

Application of this formula for each matrix size produced the results summarized in the table below. We observe an almost three-fold improvement in the number of FP operations per second in the optimized case as compared to the naïve case.

Matrix Dimension	**# FP operations**	**GFLOPs naïve**	**GFLOPs optimized**
512	6553600	14.2470	50.8031
1024	26214400	19.7546	56.7411
2048	104857600	22.0892	60.0559