PARALLEL PROGRAMMING (CO471)

ASSIGNMENT 3 - CUDA

National Institute of Technology Karnataka Surathkal

Date: 12th February, 2018

Group Members:

ROLL NUMBER	NAME	EMAIL ADDRESS
14CO151	V B Vineeth Reddy	harryvineeth@gmail.com
14CO252	Vignesh Kannan	vignesh2496@gmail.com

1)

1. Tesla Architecture, Computing Capability = 3.5
2. 512
3. 33553920 is the maximum number of threads that can be launched on the GPU.
4. The amount of co operation between the threads increases as the number of threads increases. So the performance gain obtained from using more number of threads will be neutralised by the communication cost between the threads. So we won't max number of threads in such cases. And also we can gain maximum value from parallel programming if each threads spends majority of time in computing rather than context switching.
5. The number of threads also depends upon the resources used by each thread. In case of heavy resource consumption per thread, the resource allocation can be a limiting factor.
6. 49152 is the shared memory per block.
7. Global memory is the memory allocated for device. 11995578368 is the total global memory.
8. The CUDA language makes available another kind of memory known as constant memory. As the name may indicate, we use constant memory for data that will not change over the course of a kernel execution. NVIDIA hardware provides 64KB of constant memory that it treats differently than it treats standard global memory. 65536 is the total constant memory.
9. A warp refers to a collection of 32 threads that are "woven together" and get executed in lockstep. At every line in your program, each thread in a warp executes the same instruction on different data. 32 is the warp size.
10. Yes, since the computing capability is 3.5 and double precision is supported from 3 or higher.

2)

In this program, an array of numbers needs to be summed. To do this, a fixed number of BLOCKS and THREADS_PER_BLOCK are used to form the GPU grid. Each thread in the grid is then assigned the task of computing a partial sum of the array in a Round-Robin fashion. Each block has an array in its shared memory where each thread stores its computed sum. Once all the threads in a block have finished computing their respective partial sums, the per-block array is reduced to have the sum stored in the first element of the array. This reduction is done in a divide and conquer manner. The first element of each per-block array is then stored in a result array indexed by block number. The result array is summed in the host code to compute the final sum.

3)

1. N^2 is the number of Floating point operations done by the kernel.
2. $2N^2$ is the number of Global memory reads performed by the kernel.
3. N^2 is the number of Global memory writes performed by the kernel.

4)

1. No of floating operations performed is approximately equal to (9 + 8) x 3 x N^2. As we are using a 3 x 3 kernel, each pixel in the new image requires the 9 multiplications and 8 additions per channel to compute the dot product for that channel. We have 3 channels and N^2 pixels. Hence the aforementioned cost.
2. Each thread in a block reads the R, G, B values from the 4 pixels present at the corners of its blurring neighborhood. Each read is from the global memory. We have N^2 threads where N is the width of the input square matrix. So totally there are N^2 x 4 x 3 = $12N^2$ global memory reads.
3. Each thread in a block writes to the global memory once after it has computed the average of the pixels in the neighborhood. We have N^2 threads. So there are N^2 global memory writes.
4. In the naive method of implementing the kernel, each thread in a tile (block) reads its respective 8-neighbours from the global memory to perform the averaging. But this clearly involves redundant operations. The elements retrieved by one of the threads can also be used by other adjacent threads. Also if this kind of sharing is possible, we can utilize the shared memory of a block to speed up the access by each thread in the block. Hence the following optimization can be performed. Each thread can read the 4 pixels present at the corners of its blurring neighborhood from the global memory into the shared memory of the block. Once all the threads have finished reading their respective 4 corner pixels into the shared memory (ensured using __syncthreads( )), the threads can start performing the convolution operation parallely.This is a better approach since shared memory access is much faster than global memory access.

5)

1. $5N^2$ number of Floating point operations are done by the kernel (3 multiplications and 2 additions).
2. 2D Matrix where each entry is a RGB value is better than a 3D matrix where the Z value represents the color. But in applications where we scale the colours by a different margin the 3D matrix representation works better.
3. $3N^2$ is the number of Global memory reads performed by the kernel.
4. N^2 is the number of Global memory writes performed by the kernel.
5. Using a 1d thread and 1d blocks we can achieve a better performance than when we use 2D threads and blocks.

6)

1. N^3 is the number of Floating point operations done by the kernel.
2. ($2N^3$/Block_size) = ($2N^3$/256) is the number of Global memory reads performed by the kernel.
3. N^3/Block_size is the number of Global memory writes performed by the kernel.
4. The kernel is computation bound. The current architecture of Stream Multiprocessor (SM) only allows one source operand from the shared memory. However, computing the inner product requires two source operands from from the shared memory. A better solution is to perform outer product instead of inner product. In this case, matrix A is stored in shared memory, but matrix B and C are stored in registers. The outer product does not require sharing of matrix B and matrix C, therefore, each thread only stores one element of B and one column of the tile of C in the register. The "computation-to-memory ratio" of the outer product is the same as the inner product.
5. Comparing to normal parallel multiplication program the tiled version is little complicated. In original version we calculate the values directly by accessing the global memory but in the tiled version we have two phases in first phase we gather data in shared memory and in the second phase we compute the output value.
6. In this case we can divide the matrix into 2-D grid of Blocks. Then block can compute the output value of some part of the matrix i.e we distribute the work among threads of different blocks if we exceed the maximum number of threads per block.
7. We can use some approach similar to strassen multiplication which uses divide and conquer method. Here both the matrices are divided into sub matrices of equal size and the product is calculated recursively. The size of sub matrix should be chosen such that it should fit in the global memory.

7)

1. In the first approach, every thread accesses some part of the array based on the thread index and updates the global histogram using atomic add. This approach makes the work parallel but there will be overhead because of the large number of global writes as every thread shares the same copy. To overcome this problem in second approach we created a local copy of the histogram in every block using shared memory and each thread updates the values in this local histogram and we combine all this local histograms to get final histograms, using atomic add operation. Here number of global reads will be the same but the global reads which are the overhead in the first approach are greatly reduced because we are using shared memory in this case, which is significantly faster than global writes.

2. Yes, when we use very large size of the array , all the bins in the histogram saturates and giving a value of 127. When the array size is very high like 2^32 crossing the integer value makes all the bins value zero.
3. The approach were we used multiple blocks and 256 threads in each block and each block having the local copy of the histogram to reduce the number of global writes gave the best performance.
4. No of global reads = size of the array. As we need each value of the array only once to update the histogram.
5. No of global writes = No of Blocks * 4096 . As we are using using local histogram to update the values while we are reading the values of the array,. The global array is updated only after the local histograms are created. Each block updates the global array bin size (4096) times.
6. No of atomic operations = Number of elements in the array + No of blocks * bin size. Number of elements in the array is because after reading every value in the array we update the value in local histogram using atomic add. To update the global histogram we use local histograms in each block, this is done with the help of atomic add operations contributing to another no of blocks * bin size atomic operations.
7. If all the array elements have same value it won't affect the global writes but it will affect the local writes and also the global writes, as all these writes are atomic and being performed on the element it creates a lot of overhead, as there will be lot of threads waiting to update the same value.
8. If the all the array values are random then there will equal no of elements in each bin. In this case the contention by the threads to update the value of the bin will be minimum.

8) Convolution :

1. The applications of Convolution are:
    a. Polynomial Multiplication
    b. Audio Processing
    c. Synthesized Seismographs
    d. Artificial Intelligence
    e. Optics
2. No of floating operations performed is approximately equal to (9 + 8) x 3 x N^2. As we are using a 3 x 3 kernel, each pixel in the new image requires the 9 multiplications and 8 additions per channel to compute the dot product for that channel. We have 3 channels and N^2 pixels. Hence the aforementioned cost.
3. Each thread in a block reads the R, G, B values from the 4 pixels present at the corners of its blurring neighborhood. Each read is from the global memory. We have N^2 threads where N is the width of the input square matrix. So totally there are N^2 x 4 x 3 = $12N^2$ global memory reads.

4. Each thread in a block writes to the global memory once after it has computed the average of the pixels in the neighborhood. We have N^2 threads. So there are N^2 global memory writes.
5. Since the mask size is much larger than the image size most of the time is spent in updating the shared memory. In this case the shared memory size is much larger than before. This puts a restriction on our code since the amount of shared memory per block is limited.
6. If we perform convolution in place, we will end up changing the input to the convolved value. This value will be used in further calculations of the convolution arising in errors. Since the memory used is global, it getting updated changes the output for every thread. Since the threads are parallel, we do not know what order their writes will be in.
7. The overhead for using the GPU for computation is calculated as follows:
    - Allocating space for the input image using cudaMalloc( ): cost = N^2
    - Allocating space for the kernel using cudaMalloc( ): cost = K_SIZE^2
    - Allocating space for the output image using cudaMalloc( ): cost = N^2
    - Copying the host input image into the device input image using cudaMemcpy( ): cost = N^2
    - Copying the host kernel into the device kernel using cudaMemcpy( ): cost = K_SIZE^2
    - Copying the device output image into the host output image using cudaMemcpy( ): cost = N^2

   Total cost = $3 N^2 + 3 K\_SIZE^2$

   Hence the cost scales quadratically with the input size.
8. Identity Mask is a mask which acts as an identity function when an image is convoluted with it, i.e. it gives back the input as it is.