Tutorial 3

Question 1

Consider the following reduction kernel code snippet.

```
__global__ void reduce ( int * g_idata , int * g_odata , unsigned int n ) {
extern __shared__ int sdata [];
unsigned int tid = threadIdx . x ;
unsigned int i = blockIdx . x * blockDim . x + threadIdx . x ;
sdata [ tid ] = ( i < n ) ? g_idata [ i ] : 0;
__syncthreads () ;
for(unsigned int s=1; s < blockDim.x; s *= 2)
{
int index = 2 * s * tid;
if (index < blockDim.x)
sdata [ index ] += sdata [ index + s ];
__syncthreads () ;
}
if ( tid == 0)
g_odata [ blockIdx . x ] = sdata [0];
}
```

Which of the following option is correct?

A. The kernel suffers only from high divergence.
B. The kernel suffers only from shared memory bank conflicts.
C. The kernel suffers from high divergence as well as memory bank conflicts.
D. None of the above

Answer: B
Solution: Refer to slides for Reduction 2 kernel

Question 2

For the following code snippet for reduction, find the total number of memory write accesses between line number 6 and 10 per block. The blockDim.x is 64 and warp size is 16.

Kernel

```
1. unsigned int tid = threadIdx .x;
2. unsigned int i = blockIdx .x* blockDim .x + threadIdx .x;
3. __shared__ float sdata[64];
4. sdata [ tid ] = (i < n) ? g_idata [i] : 0;
5. __syncthreads ();
6. for (unsigned int s=blockDim.x/2; s>0; s»=1){
7. if (tid < s)
8. sdata [ tid ] += sdata [tid + s];
9. __syncthreads () ;
10. }
```

Number of write accesses :

A. 10
B. 5
C. 7
D. 20

Answer: C

Solution:

For each possible value of s, number of write accesses :
s = 32: the threads write elements 0–31 . That covers two 16-element blocks, so there are 2 write transactions.
s = 16: the threads write elements 0–15 .which fits into one 16-element block, so there are 1 write transaction.
s = 8: the threads write elements 0–7, still inside one 16-element block, so there is 1 transaction..
s = 4:the threads write elements 0–3, again inside one block, so there is 1 transaction.
s = 2: the threads write elements 0–2, again inside one block, so there is 1 transaction.
s = 1: the single active thread writes element 0, so there is 1 transaction.
Total number of write accesses per block = 2+1+1+1+1+1=7.

Question 3

Referring to the above code snippet find the total number of memory read accesses between line number 6 and 10 per block as described in question 2.

Number of read accesses :
A)10
B) 20
C) 23
D) 30
Answer: A
Solution:

For each possible value of s, number of read accesses :
s = 32: the threads read elements 0–31 and 32–63. That covers four 16-element blocks, so there are 4 read transactions.
s = 16: the threads read elements 0–15 and 16–31. That covers two 16-element blocks, so there are 2 read transactions.
s = 8: the threads read elements 0–15, which fits into one 16-element block, so there is 1 transaction.
s = 4: the threads read elements 0–7, still inside one 16-element block, so there is 1 transaction.
s = 2: the threads read elements 0–3, again inside one block, so there is 1 transaction.
s = 1: the single active thread reads elements 0 and 1, both in the same block, so there is 1 transaction.
Total number of read accesses per block = 4+2+1+1+1+1=10.

Question 4

Consider the most naive version of the reduction kernel being launched with the number of threads in a block as 1024 on an array of size 2^22. How many times should the reduction kernel be invoked from the host program?

A. 3
B. 5
C. 7
D. 9

Answer: A

Solution. In the first iteration 2^22 elements get reduced to 2^22/2^10=2^12. In the second iteration, 2^12 gets reduced to 2^12/2^10=2^2. In the third iteration, the reduction operation completes.

Question 5:

Consider the version of the reduction kernel where the first addition operation occurs during the initial global load kernel being launched with the number of threads in a block as 32 on an array of size 2^12. How many thread blocks are launched in the first invocation of the kernel from the host program?

A. 32
B. 64
C. 128
D. 16

Answer: B

Solution: If the number of elements to be reduced is n, then a total of n/2 threads are launched for the given reduction kernel call. Given this, in the first iteration we have 2^11 threads for 2^12 elements. Since we have 32=2^5 threads per block, a total of 2^11/2^5=2^6 blocks are launched.

Question 6:

In a CUDA reduction kernel, if the input array size is n=1024 and threads per block are 32, how many total threads are required in the second kernel invocation, assuming two elements are processed per thread in the first iteration?

A. 32
B. 64
C. 128
D. 256

Answer: D

Solution:
First iteration: n/2=ceil(1024/2)=512 threads required.
Threads per block: 32.
Second iteration: Further reduces threads to 512/2=256.

Question 7:

Consider the following CUDA kernel code

```
__global__ void kernel_reduce(const float* const d_array, float* d_max, const size_t elements){

    extern __shared__ float shared[];
    int tid = threadIdx.x;
    int gid = (blockDim.x * blockIdx.x) + tid;
    shared[tid] = -FLOAT_MAX;

    if (gid < elements)
```

```
    shared[tid] = d_array[gid];
  __syncthreads();

  for (unsigned int s=blockDim.x/2; s>0; s>>=1)
  {
    if (tid < s && gid < elements)
      shared[tid] = max(shared[tid], shared[tid + s]);
    __syncthreads();
  }
  if (tid == 0)
    d_max[blockIdx.x] = shared[tid];
}
```

The above kernel computes:

A. Sum of array elements using reduction
B. Minimum of array elements using reduction
C. Maximum of array elements using reduction
D. Multiplication of array elements using reduction

Answer: C

Question 8:

Analyze the following code snippet for divergence:

```
int idx = threadIdx.x;
if (idx % 2 == 0) {
   arr[idx] = idx * 2;
} else {
   arr[idx] = idx * 3;
}
```

Which statement best explains the behavior of this code regarding thread divergence?
A. Divergence occurs because threads within a wrap follow different paths based on their index
B. No divergence occurs because idx % 2 is a uniform conditional
C. Wrap size ensures even and odd threads are separated
D. Wrap execution is independent of branching

Answer: A

Solution:

Divergence occurs because threads in a warp take different paths (even vs odd).

Question 9:
In a GPU system with memory transaction width of N, a global memory access can coalesce (bring in a single transaction) N consecutive floating point data values. Consider the following code snippet.

```
__global__ void mem_access(float* A)
{
        int tid = threadIdx.x;
        for(int i=1; i<=32; i=i*2)
                A[tid*i]+=2;
}
```

Let the number of threads be 256 (tid = 0 to 255) and the size of the array A be 8192. Assuming a warp size of 16, compute the total number of global memory transactions made in the for loop for transaction widths of 16 elements. Assume no caching occurs.

A. 752
B. 992
C. 1504
D. 1024

Answer: C
Solution:
Given the fact that the total number of threads is 256 and the warp size is 16, the total number of warps is 16. For every transaction width, the total number of memory transactions will be the sumtotal of the number of memory transactions in each iteration which are as follows

Iteration 1: Array elements 0 to 256 are accessed. Warp of size 16 threads accesses 16 consecutive elements, which lie in one block. So, each wrap accesses one 16-element block.

Transactions per warp = 1 read+1 write=2

Total transactions = 16 warps×2=32

Iteration 2: Array elements 0,2,4,...,512 are accessed. (256 threads, stride = 2).Within a warp, 16 threads touch 16 even indices spread across 2 consecutive 16-element blocks. So, each warp accesses two such blocks.

Transactions per warp = 2 read+2 write=4

Total transactions = 16 warps×4=64

Iteration 3: Array elements 0,4,8,... 1024 are accessed. (stride = 4).Within a warp, 16 threads spread their indices across 4 different 16-element blocks. So, each warp accesses four such blocks.

Transactions per warp = 4 read+4 write=8

Total transactions = 16 warps×8=128

Iteration 4: Array elements 0,8,16,... 2048 are accessed. (stride = 8).Within a warp, 16 threads span 8 different 16-element blocks. So, each warp accesses eight such blocks.

Transactions per warp = 8 read+8 write=16

Total transactions = 16 warps×16=256

Iteration 5: Array elements 0,8,16,... 2048 are accessed.(stride = 16). Within a warp, 16 threads fall into 16 different 16-element blocks (each thread in its own block). So, each warp accesses sixteen such blocks.

Transactions per warp = 16 read+16 write=32

Total transactions = 16 warps×32=512

Iteration 6: Array elements 0,16,32... 8160 are accessed.(stride = 32).Within a warp, 16 threads again map into 16 distinct 16-element blocks (each thread separate). So, each warp accesses sixteen such blocks.

Transactions per warp = 16 read+16 write=32

Total transactions = 16 warps×32=512

The total number of memory transactions for warp size of 16 and transaction width 16 will be 32+64+128+256+512+512=1504.

Question 10:

Assume there is a 1D input array A of N integers, where each element is of the form A[i]=i, where i = 0 to N-1. Consider the following kernel code snippet executing on a GPU architecture where the warp size is 8.

```
__global__ void divBranch(int* A, int* B, int M, int N, int k)
{
    int tid = threadIdx.x;
    if(A[tid]%8)
        B[tid]+=sqrt(A[tid]);
    else if(A[tid]%4)
        {
            B[tid]+=sqrt(A[tid]);
            B[tid]+=sqrt(A[tid]);
        }
```

```
    else{
                B[tid]+=sqrt(A[tid]);
                B[tid]+=sqrt(A[tid]);
                B[tid]+=sqrt(A[tid]);
                B[tid]+=sqrt(A[tid]);
        }
}
```

The above code represents a kernel suffering from divergence. During the lifetime of warp 0 what are the total number of square root instructions that are executed ? Choose from the correct option below.

A. 0
B. 15
C. 29
D. 11

Answer: D

Solution: For warp 0, only thread id 0 does not satisfy the first if condition while thread ids 1-7 do. Also the second if condition is not satisfied by thread id 0. Thus the total number of square root instructions executed by threads in warp 0 = 1*4 + 7*1 = 11

Assignment 1

Implement the following host, kernel programs and execute on an NVIDIA GPU. 2D convolution is primarily used in image processing for tasks such as image enhancing, blurring, etc. Let us consider 2D convolution operation for a 2D Matrix of floating point numbers. It is a neighborhood operation where each output element in the output matrix (OUT) is the weighted sum of a collection of neighboring elements of the input matrix (IN). The weights used in the weighted sum are typically stored in an array called the convolution mask or filter (M). The convolution formula for a 3x3 convolution filter M with a matrix IN of size nxn is:

$$P(i,j) = \sum_{a=0}^{2}(\sum_{b=0}^{2}(M[a][b]*IN[i+a-1][j+b-1])) \text{ where } 0 \leq i \leq n \text{ and } 0 \leq j \leq n$$

For Example.
Input
n=3
IN
3.0 3.0 3.0
3.0 3.0 3.0
3.0 3.0 3.0

M
 0 -1 0
-1 5 -1
 0 -1 0
OUT
9.0 6.0 9.0
6.0 3.0 6.0
9.0 6.0 9.0

Note for boundary elements, a padding of 0.0 is assumed. In the above example, for element input[0][0], placing the 3x3 mask with its centre at IN[0][0], produces OUT[0][0] = 0.0*0+0.0*-1 +0.0*0 + 0.0*-1 + IN[0][0]*5 + IN[0][1]*-1 + 0.0*0 + IN[1][0]*-1 +IN[1][1]*0 =0.0+0.0+0.0+0.0+15.0-3.0+0.0-3.0+0.0 = 9.0.

Implement a CUDA program which takes as input, two variables n and m, and a matrix N where

i) n = argv[1] is the width of the input square matrix
ii) m = argv[2] is the width of the square mask matrix (m must be odd).
iii) N is the input matrix stored in a text file, with n*n lines, each line having one element of the matrix in a row major order.

You can initialize the mask internally in your program.

The output will be an NxN matrix where each element in a row is written in a new line in the row major order.

TODO: Submit a complete archive to the TA in his email id by 25th August 9pm. Code should not be AI generated, code should not be plagiarised. If caught, you will be banned from all submissions.

Profile the program using NVIDIA profiling tools to determine the number of global memory load transactions and the total memory consumption (in bytes), and present these results along with the program's output

Report various classes of operations (branch, arithmetic, memory etc) and their relative time taken. Provide the data, charts etc in a separate report. In this report also justify whether using shared memory will help in this program.

Put both report and code in the archive along with EXACT commandline instructions on how to run your code and reproduce the profiling data.

Refer to the following input output example format.

argv[1] = 3
argv[2] = 3

Input.txt
3.0
3.0
3.0
3.0
3.0
3.0
3.0
3.0
3.0

Output.txt
9.0
6.0
9.0

Solution:

Matrix size NxN
Mask Size MxM
Tile Size MxM

Naive Algorithm-

Total Mem. accesses required= NxN x MxM

Shared memory Algorithm-

Mem. accesses for computing a tile = (Mem. accesses to load a tile and boundary elements) = (M+2)x(M+2)
Total Mem. Accesses = (Mem. accesses for computing a tile) x (No of Tiles)
= (M+2)x(M+2) x (N^2/M^2) = NxN x (MxM +4*M + 4)/MxM

Assignment -2

Consider the various reduction optimizations taught in class, from reduction1 up to reduction5.

Since reduction5 considers only two global loads per thread, call it reduction5.1. Let us say reduction5.2 is the variant where 3 global loads are considered per thread, similarly reduction5.n is the variant with (n+1) global loads.

With input data size varying from 2^16 to 2^32, check the execution time for reduction operation with each reduction code from reduction0 to reduction5.n (for n = 1 to 8).

Submit an archive with

1. all individual reduction codes (host+device in each case).
2. a script that automatically runs them and generates the graph similar to slide 45.
3. A word doc with interesting observations on the profiling data about operation mix, memory vs compute, effect of optimization at each step etc in each case.

DO not copy or use GPT (we shall check for plagiarism and AI content both)
consider reduction operations other than addition : min, max, average, and, or
etc